

University of Edinburgh

School of Informatics

A Visual Single-Step Navigator for the Eclipse PEPA Plug-in

4th Year Project Report
Computer Science

Paul McEwan (0452900)

March 3, 2008

Abstract: The Performance Evaluation Process Algebra (PEPA) language allows model systems to be created and evaluated. Previously, the main tool used to work with PEPA files was the PEPA Workbench, which contained - among many features - a visual debugger. The debugger created a visual, state machine representation of the PEPA file and allowed users to navigate through different states and perform simulation runs. The Workbench is slowly being replaced by a new PEPA Plug-in for Eclipse, utilising newer technology and hopefully exposing the benefits of model systems to a larger number of users. This project aims to provide a Visual Single-Step Navigator for the Eclipse PEPA Plug-in.

Acknowledgments

I would like to say thank you to the following:

- Stephen Gilmore: my project supervisor, always ready to answer my questions and to meet up
- Mirco Tribastone: for his efforts in helping me understand the Eclipse PEPA Plug-in API
- Ian Bull: the author of the Zest graphing library used to create the visual representation of the PEPA models
- My Father: for his proof-reading skills and his willingness to listen to me discuss problems about activity arcs, process nodes and PEPA files
- All my family and friends: for your love and support through the ups and downs

Contents

1	Introduction	1
1.1	PEPA	1
1.2	PEPA Workbench	4
1.2.1	Visual Single-Step Debugger	4
1.3	Eclipse PEPA Plug-in	7
1.4	Goals	7
2	Design	11
2.1	Eclipse PEPA Plug-in API	11
2.1.1	Information on the System Modelled	11
2.1.2	Information on Current System State	12
2.2	Visualisation	14
2.2.1	The Visual Representation of the PEPA Model	14
2.2.2	Technology used to create the Visual Navigator	15
2.2.3	Interface Design	19
3	Implementation	21
3.1	Java Representation of the PEPA Model	21
3.1.1	Rate	22
3.1.2	PEPAComponent	22
3.1.3	SystemEquation	23
3.2	Obtaining the Data about PEPA Model	24
3.2.1	Instantiating the PEPAModel Object	25
3.2.2	State Space and State Information	30
3.2.3	Changing State and Updating	32
3.3	Visual Navigator	34
3.3.1	Graph Representation of PEPA Model	34
3.3.2	Single-Step Mode	36
3.3.3	Simulation Mode	42
4	Evaluation	49
4.1	Comparison with the Visual Single-Step Debugger	49
4.1.1	How They Represent the model	49
4.1.2	Features	50
4.1.3	Overall	52
4.2	Testing	52
4.2.1	System Testing	52
4.2.2	User Testing	58
4.3	Goals Met	59

4.4	Limitations	60
4.5	Future Improvements	62
5	Conclusion	65
	Bibliography	67
	Appendices	67
A	Algorithms	69

1. Introduction

The design of large, expensive computer systems requires careful thought and planning. Systems which do not behave as expected - or perform the task in an unacceptable time - can add additional cost and time to a project. This can result in problems using the system once built and deployed or, in the worst case, endanger lives. The modelling of the system prior to construction provides the ability to evaluate the system performance and identify potential problems early, by performing simulated runs and experimentations.

When creating a model of a system, it is quite common to model it as a *discrete event* system [8]. In a discrete event system, the system is represented as a collection of variables that take specific values, with a change in value occurring as part of an event. This type of model can then be used in a *discrete event simulation*; a simulated run of the system represented by a series of events occurring in some order [2]. A number of performance modelling paradigms use this concept, including *Stochastic Petri Nets* (SPN) and *Queueing Theory* [8]. A visual state-machine representation of models created using either of these paradigms is relatively straightforward and can help in understanding the systems modelled.

1.1 PEPA

Another popular performance modelling paradigm which uses discrete event systems and simulations is the Stochastic Process Algebra (SPA)[1]. This formalism allows the modelling of systems in a high-level language. The models encapsulate the system in terms of processes and activities; each activity having an exponential delay attached to it. These systems can have their performance evaluated in a temporal and/or probabilistic fashion. If a probabilistic analysis is required, the probability of reaching states of the system are calculated using Markov processes. There are several unique features of SPA:

- *Compositionality*: model a system as a series of subsystems that interact with one another.
- *Formality*: every term used in the language has an exact meaning.
- *Abstraction*: the internal functioning of a system does not need to be considered even if using detailed components to create models that are complex.

There are many languages which use SPA to model systems [1]. The language used as part of this project shall be PEPA - the Performance Evaluation Process

```

lambda = 2.0
omega = 2.0
mu = 2.0

PC10 = (arrive1,lambda).PC11 + (walkon2,infty).PC10
PC11 = (serve1,infty).PC10

PC20 = (arrive2,lambda).PC21 + (walkon1,infty).PC20
PC21 = (serve2,infty).PC20

S1 = (walkon2,omega).S2 + (serve1,mu).(walk2,omega).S2
S2 = (walkon1,omega).S1 + (serve2,mu).(walk1,omega).S1

(PC10 <> PC20 ) <walkon1,walkon2, serve1,serve2> S1

```

Figure 1.1: *An example PEPA file illustrating a system with 2 computers and a token ring server.*

Algebra language. The language contains all the features found in process algebra and adds additional features. For example, to further emphasize the relation between the model and a Markov process, each activity is assigned an exponentially distributed variable representing the duration of the activity. An example PEPA file is presented in Figure 1.1. This example uses PEPA syntax to illustrate a PC-LAN system with two computers and a token server, allowing each computer to send data.

A PEPA model consists of one or more components, which may interact/co-operate with each other or may perform individually but in parallel. These components have actions attached to them, the activities having a type and a parameter describing its exponentially distributed duration. To allow the creation of more complex systems, PEPA provides a set of combinators:

- Prefix (.)

The prefix operator can be used in two situations: a component can perform an activity to progress into a new state or return to its current state; or, a sequence of activities can be performed one after the other before, eventually, entering a specific state. An example of the prefix operator from the PCLAN-2 example would be: $PC11 = (serve1, infty).PC10$. Here, the component is in a state where it has data to send; performing the serve activity causes the component to return to the initial state (PC10) where it does not have anything to send.

- Choice (+)

The choice operator allows a component to have more than one option for activities to perform, each taking the system into a different state. For *PCLAN-2*: $PC10 = (arrive1,lambda).PC11 + (walkon2,infty).PC10$. In this example, the component has two choices, representing alternative actions the component may take: the component could receive data which it wishes to send, in which case it moves into the state of having data; or, the activity signalling the token server to move on to the second PC could occur, meaning this PC has no data to send and therefore returns to its current state.

- Co-operation ($\langle \rangle$)

The co-operation operator is found at the end of a PEPA file, inside the System Equation. The operator describes the relationship between components, stating any activities the components co-operate or synchronise on. If the two components act completely independently of each other, then this is represented as “ $\langle \rangle$ ”, meaning there are no activities they co-operate on (the symbol “ \parallel ” may also be used for this purpose). If some components co-operate on an activity “act”, this is represented as “ $\langle act \rangle$ ”, meaning this action can only be performed when the system is in a state allowing all synchronising components to perform the activity.

Using these models, the state space (all the possible configurations of the system) can be generated, with a steady-state probability being evaluated for each of these states. Using this information, useful information such as utilisation and throughput of the system can be obtained. In addition, it is possible to use these models to perform experiments, examining how the system behaves under different circumstances.

When modelling real world systems, multiple instances of components may exist as part of the system. For example, a computer network system will have multiple machines and possibly multiple servers. Creating multiple instances of these objects can quickly lead to a state-space explosion, resulting from multiple configurations for each component. To avoid this problem, PEPA employs the technique of “aggregation”. The technique is best illustrated by the diagram in Figure. 1.2. Essentially, PEPA allows the grouping of similar components into a single group. For example, if a system has a “Browser” component, 3 browsers could be represented as “Browser[3]”. Grouping in this manner means that situations in which 1 or 2 browsers are being used are considered only once, reducing the number of states produced in the state space.

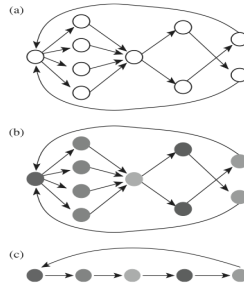


Figure 1.2: *An example of aggregation. When viewed as columns, the four nodes in the second column have the same previous and next nodes (b) and so are condensed easily into one node representing the group of four (c). The two nodes of the fourth column have the same previous node but different next nodes (b). The nodes in the fifth column have different origin nodes but the same destination node (b). Since in aggregation an explicit interpretation about the state is not required, the fourth and fifth columns can be grouped together, with the original and destination node for the fourth node being the third and fifth respectively, while the fifth node has the fourth as the origin and the first as the destination (c). Picture from [1].*

1.2 PEPA Workbench

Although the use of PEPA and SPA has not been widespread outside academic circles, there have been tools developed to make its use more widespread and common. The “PEPA Workbench” is probably the most well known. Originally written in the functional programming language Standard ML, the current version is a standalone Java application. The PEPA Workbench allows the creation and manipulation of performance models via “.pepa” files. The PEPA model’s state-space can be derived, allowing the designer to study the steady-state and state-output of the systems, among other things. Using the model, the designer may also conduct experiments, single-step debugging (choosing next state) and much more. In its current form, the PEPA Workbench does not support the use of aggregation in PEPA files.

1.2.1 Visual Single-Step Debugger

The “Visual Single Step Debugger” is a useful tool found in the workbench for use by experts and novices alike, and was the subject of a previous Honours project [2]. The visual debugger takes the PEPA model and generates a graphical representation of the system with graphs representing the components, where the nodes represent activities and processes and the vertices represent the transitions

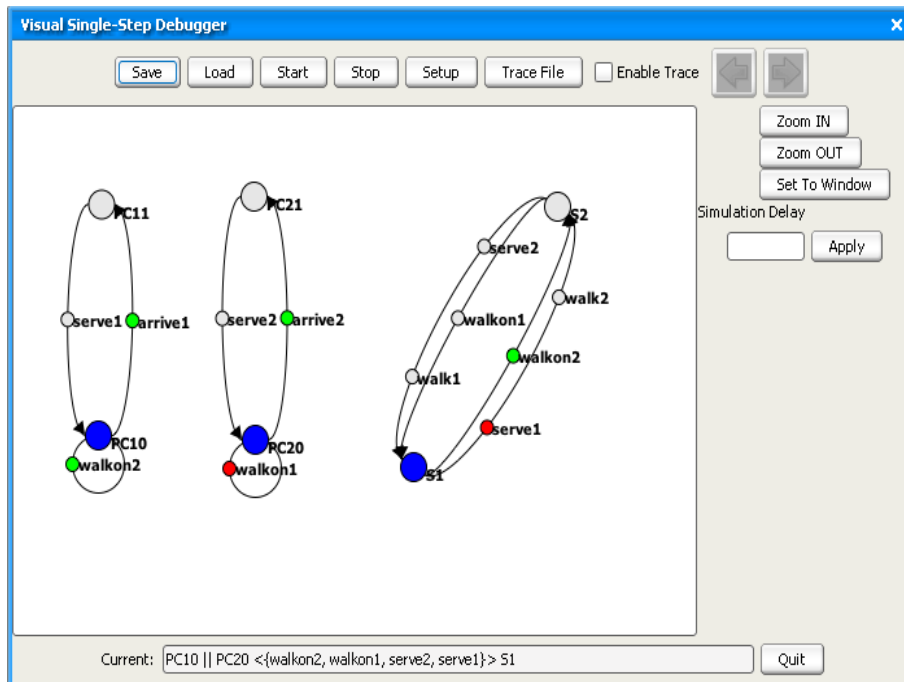


Figure 1.3: *The PC-LAN2 example PEPA file visualised using the visual debugger*

possible for each processes. A screen shot of the visual debugger with the PC-LAN2 code is presented in Figure 1.3. Each node is coloured to represent the state that particular node is in: blue if the node is currently occupied; green for an activity node whose action can be fired; red for activities that could be next but are prevented from doing so due to co-operation issues; grey for those nodes that are inactive in that state.

As previously stated, the visualisation tool for PEPA models was popular among many PEPA users. The ability to see the system represented visually as a graph of nodes and vertices makes not only understanding the system and what is happening simpler, but allows for a more intuitive representation for novices.

This visual debugger allows the user not only to visualise their model and manipulate the presentation, but provides the ability to navigate through the system states via single steps (similar to the textual single step debugger). The debugger allows the user to select which activity should fire by clicking on an activity node that is coloured green: this performs the corresponding activity and sends the system into a new state. The visual debugger also provides a simulation mode to perform a simulated run of the system, with the states visited based upon the activity rates.

Although extremely useful, the visual debugger does exhibit a few limitations:

- Multiple instances of the same component cannot be represented. For ex-

ample, in the PC-LAN2 example from before, a user may wish to model a system in which there were two token ring servers running concurrently. This would be represented in the PEPA file through altering the System Equation to include an additional instance of the token ring server (either “S1<>S1” or “S1[2]”). Unfortunately, the visual debugger cannot represent multiple components by using “S1<>S1”, instead only creating one instance, and does not have support for the use of aggregation.

- The representation of both activities and processes as nodes can often lead to a large number of nodes being created and presented on screen. Furthermore, when a sequence of actions must take place to go from one process state to the other (e.g. “P = (a, r).(b, r).Q”), activity nodes are used as “current state” nodes and coloured blue (in this example, the activity node representing “(b, r)” would turn blue if the ‘a’ activity was fired). This can cause problems for two reasons, the first and most obvious one being that if the node is blue there is no way to know if it can fire or not (the green, red or grey colour cannot be seen). The second problem lies in the interpretation of the graphs: the activity nodes should simply represent actions that can take place at a given instance. Using these nodes to represent both actions that can take place and states of the machines can lead to a misinterpretation of the PEPA models.

A slightly more accurate representation would involve using “intermediate nodes”: nodes that can be entered but represent an intermediate step in the transition from one process to the other. However, the inclusion of intermediate nodes would increase the total number of nodes in the graphs. This could be remedied by removing the activity nodes and simply using a labelled arc. This arc could then be highlighted red, green or grey, while the process and intermediate nodes need only be blue or grey.

- The graphs can be manipulated in one of three ways: by clicking on an activity node to fire an activity; clicking and holding on a process node to drag it around; clicking and holding on the graph to move the whole graph. These functions are given to the left, right and middle mouse buttons respectively. However, this means that all the features of the interface can only be used by a three-buttoned mouse. In addition, requiring the use of the right and middle buttons means these features may be “hidden” from some users.

1.3 Eclipse PEPA Plug-in

The PEPA Workbench has been around for a number of years now, but is slowly being phased out in favour of a new tool. The PEPA Plug-in for Eclipse takes the functionality of the workbench and, utilising newer technology, integrates it into the popular Integrated Development Environment (IDE) Eclipse.

The Eclipse IDE is considered a fairly friendly IDE for novices while having the power suitable for expert programmers, making it a commonly used and familiar environment for many in computing. It is hoped the PEPA Plug-in for Eclipse will not only allow easy access and use for those experienced with stochastic modelling, but will expose the benefits of this design and planning paradigm to many who would otherwise not know about it. The current version of the plug-in implements many of the features contained in the Workbench, such as the ability to create models, perform textual single-step debugging, perform experiments and graph the performance of the system (such as throughput). In addition, the Plug-in includes support for aggregation, a feature missing from the workbench.

Despite all the features implemented so far, one key feature of the Workbench has not yet been implemented: the visual debugger. As stated previously, this tool was popular with PEPA users, particularly novices and first time users. Considering the fact that the Eclipse Plug-in was developed in-part to attract new users, the inclusion of a visual debugger is a must.

1.4 Goals

The goal of this project is to create a visual single step navigator for the Eclipse PEPA Plug-in. This visual navigator should have a similar appearance and functionality to that of the PEPA Workbench visual debugger, including:

- A representation of each component of the PEPA model using a graph with nodes and vertices illustrating the processes and activities
- The colour scheme used for the debugger should be both re-used and retain the original meaning (blue for current state, green for an activity that can fire etc.)
- Allowing the user to transition between states by clicking on an activity highlighted green on the graph, thereby performing single-step navigation
- Allowing the user to move backwards/forwards through previously visited states

- Being able to zoom in and out, allowing particular component graphs to be more clearly visible
- Providing a simulation mode, which performs automatic single-step navigation based upon some probability for selecting which activity to fire. The simulation mode should also allow the user to:
 - set time limits
 - time between transition fires
 - the generation of trace files

In addition, the new visual navigator should include some additional features

- *Intermediate Nodes*

Intermediate nodes represent states of the system when a transition from one process to the next has not been completed, as a result of multiple actions needing to fire in sequence. These intermediate states are still representative of a PEPA model and, more importantly, would provide a clearer visual representation of the state of the transition.

- *Activity Arcs*

The addition of labelled activity arcs would provide less clutter from numerous nodes as well as create a clearer distinction between the process and intermediate nodes (states of the components) and the activities (actions that, when fired, change the state of a component). These arcs would be coloured to represent the activity's state.

- *Aggregation mode*

The inclusion of aggregation will allow multiple instances of a component to be represented without requiring multiple graphs to be drawn. Using aggregation instead of multiple instances of a component in the System Equation is also considered good PEPA coding, as it reduces the size of the derived state-space.

- *Highlight mode*

Providing the user with some visual indication regarding the history of the transitions made in the navigator. This could include highlighting states the system has been in before; highlighting nodes that have previously been current; highlighting activity arcs that have been previously fired. This could help identify states that are never visited or activities that cannot ever be fired

- *Visual Representation of the System Equation*

An important part of the PEPA model, a visual representation of the System Equation that the user could load when needed may be useful. The visual System Equation would allow easy identification of how the components co-operate with one another and on which activities (if any).

As an additional note, the new visual navigator will use the PEPA API designed for the Eclipse Plug-in, an API different from the one used in the Workbench (and therefore the original visual debugger). Although this difference will not be apparent to the end user, this change of API necessitates an additional project goal of being able to create a visual navigator using the newer API.

2. Design

During the design phase, two key components were identified. The first was the component required to retrieve information about the PEPA model from a file and obtain information on the states of the system. The second was the component required to visually present the PEPA model to the user, including what visual style to use as well as what technologies to use to create the visuals.

2.1 Eclipse PEPA Plug-in API

The API for the Eclipse PEPA Plug-in provides all the code needed to create, understand and use PEPA models, such as calculating state space, state space probabilities and utilisation. The package of most importance for this project was the “uk.ac.ed.inf.pepa” package, containing methods that could read in PEPA files and calculate the states of the model.

2.1.1 Information on the System Modelled

Using the classes in the package, a PEPA model can be read in from a “.pepa” file and parsed to interpret the parts of the model. The model is represented internally in the form of an Abstract Syntax Tree (AST). The AST contains four top level nodes: RATES, PROCESSES, System Equation and PROBLEMS:

- RATES

This part of the tree describes the user defined rates for actions. The node has any number of “Rate definition” child nodes, which describes a rate as “Name” and “Value” children; the children of which corresponds to the name of the rate and its value accordingly.

- PROCESSES

This node has one or more “Process definition” child nodes, which describes all the processes defined in the PEPA model. Each Process definition has a “Name” child, process name, and a “Process” child, which describes the associated process equation. A Process could have a “Choice” child or a “Prefix”.

The Prefix node describes the activity that can be performed and the target. The “Activity” has only one child, the activity that leads to the target (the associated activity name and rate are included as children). The “Target”

node may have just a constant, name of the process activity leads to, or another Prefix node with another activity and target, identifying a sequence of activities.

The Choice node allows a left and right hand side that contains a Prefix, representing the ability to choose from different course of actions.

- System Equation

The child of the system equation is a “Cooperation” node. The Cooperation node has children for the left hand side, right hand side and the activities the two sides cooperate on. The left and right hand sides can either have a “Constant” child - the process component involved in the cooperation - or another “Cooperation”, to represent more than two components in the system equation.

Aggregation information can be obtained from the “Cooperation” node. If a component has had an aggregation value assigned to it, the child node of the “Cooperation” will be an “Aggregation” node instead of “Constant”. This node type has two children: “Process”, the process to which it relates (the child of which is a constant that leads to the name of the process); “Copies”, the number of aggregated copies of this component.

- PROBLEMS

This node describes any problems in the PEPA model. These list problems that the plug-in has detected by analysing the file, such as definitions that have been declared but not actually used. Some of these problems are warnings (such as the unused variable), which are not the result of incorrect PEPA syntax. If a file contains only warnings, it is still a valid PEPA model. Some problems might however be errors (e.g. incorrect syntax used). If the file contains these, the PEPA model is incorrect and operations, such as deriving the state space, cannot be performed until the problems have been addressed.

Traversal of the AST allows the structure of the model, including the parts of the model - the process, activity and transition information - to be obtained. The information obtained can then be stored as Java objects and used to draw the visual representation of the PEPA model.

2.1.2 Information on Current System State

Once the PEPA model structure has been analysed and stored in a suitable format, the next key step for the single-step navigator is to obtain information

about the current state, work out what activities can fire in this state and what state the machine enters when the activity fires.

Using methods provided by “uk.ac.ed.inf.pepa.ctmc.derivation.IStateSpace”, it is possible to get information about the state space of the model. Using a number as an index, the state associated with that index can be retrieved. Using the obtained state, it is possible to find out which processes are current and which states can be outgoing (i.e. next) states from the current one. The outgoing states are represented as a list of numbers (their state space index).

With the index of any two states, the activity that must fire to move from the first state to the next can be obtained. Using this technique allows for identification of what activities may be fired: given the current state and the outgoing states, applying the method using the current state and each next state, it is possible to find out all the activities that can fire. Those activities that are not returned from the method cannot be fired and so are either blocked (it is possible to fire the activity from this process, but synchronisation prevents it) or blank (if not related to current state of the system).

This means that, as long as a variable is used to store the index of the current state the machine is in, it is possible to generate information about how the system looks, what activities can/cannot fire and to which state the activity fired will take the system.

The knowledge of the index of the current system state could also be used for highlight mode. When entering a state, the index of the state could be checked against a list containing the indices of the states previously visited. If the current states index is not in the list, then it is added. If it is in the list, highlight mode can then change the visual appearance to represent re-entering a previously visited state.

As a final point for this section, the state space information can be used to identify intermediate states and obtain the correct aggregation value for processes.

- Intermediate states are represented in the state space by a concatenation of activities still required to be fired followed by the destination process (e.g. “(a,r).(b,s).Q”). By assigning the intermediate nodes a name that matches this representation of an intermediate state, the identification of when a component is in a particular intermediate state - and therefore the corresponding intermediate node should be set to current - would be straight forward.
- As mentioned previously, with the given index of a system state, it is possible to find out which state of the component is current, and therefore which node should be set as current. This information is represented either as the name of the process/intermediate state or, if aggregation is used, poten-

tially a list of these names. When aggregation is used, beside the name is a number inside square brackets representing the aggregation value of that item in the component for the given state. This number could be extracted and used to assign the aggregation value for the corresponding node on the graph.

2.2 Visualisation

When deciding how to visually present the PEPA models to the user, three separate components required consideration. The first was the visual representation of the PEPA model desired (i.e. how the components and system equation should look visually). The second consideration was for the technology used to implement the visualisation of the PEPA models, as well as the interface for the visual navigator. The third consideration was the design of the interface, how the interface would look and how the functionality would work

2.2.1 The Visual Representation of the PEPA Model

When deciding how to visually represent the components of a PEPA model, the initial logical consideration was to have a visual representation similar to that of the Visual Debugger's representation. In this, both processes and activities were represented as nodes, with each component represented as a separate graph comprising these nodes connected by edges. As stated previously however, this representation was to be altered slightly for the new Visual Navigator. Components of the PEPA model would still be represented by separate graphs comprising nodes and vertices, but now the vertices would be labelled and represent the activities. These vertices would connect between 2 process nodes (or intermediate nodes if a sequence of actions needs to be performed). An example of this new representation is illustrated in Figure 2.1. Note that, as in the original Visual Debugger's representation of the PEPA model, activity arcs have an arrow head to further identify the result of performing the action.

The visual representation of the PEPA model reuses the colour scheme of the Workbench Visual Debugger, as the colours chosen were intuitive and will aid those familiar with the current Visual Debugger to understand the new Visual Navigator. The process and intermediate nodes will be blue if the component is currently in that state ("current node"), otherwise grey. The activity arcs will be either green (an activity that can fire from the current node), red (the activity from the current cannot fire due to co-operation requirements) or grey.

Having made the decision on how to implement the features that were already

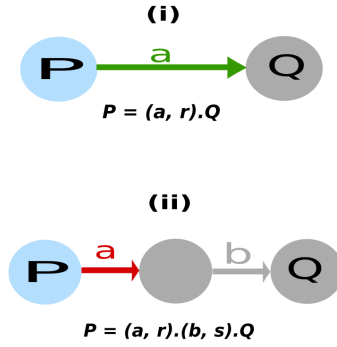


Figure 2.1: *Initial design concept for new representation of PEPA components. (i) a simple transition from one process to another that can fire next. (ii) a transition involving an intermediate node, as two actions must fire to move from one process node to the next. This first activity is blocked.*

included in the original debugger, the next design consideration involved deciding how to visually represent the features of the PEPA model that were to be new for the Visual Navigator.

- *Aggregation mode* would best be represented by a number within square brackets that contained the aggregated value for that process or intermediate step. This number could then be placed inside the respective node.
- For the *Highlight Mode*, the initial decision made was to highlight visually process and intermediate nodes which had been visited by having a yellow “halo” placed around them, while activities that had been fired previously would use green, red or grey with a slightly different tint to distinguish them.
- The visual representation of the *System Equation* which was conceived is illustrated in Figure 2.2. The components and co-operation lists of the System Equation are represented by nodes. The colour used for the component and co-operation nodes are different for easy distinguishing. The components are connected to the others via the co-operation activities in a hierarchical manner. For the visual representation, a tree like notation was chosen, where the left and right branches co-operate with one another over the activities listed in the parent co-operation node. The root node is always a co-operation, while the leaf nodes are always components.

2.2.2 Technology used to create the Visual Navigator

After the approach to represent the PEPA models had been decided, the next step was to consider what technologies and libraries were to be used to implement

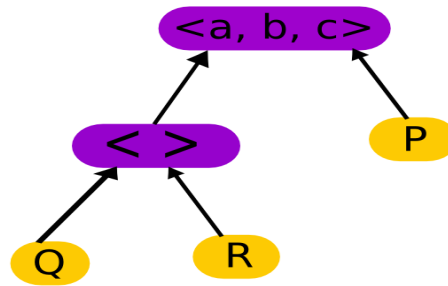


Figure 2.2: *Visual representation of the System Equation “ $(Q \langle \rangle R) \langle a, b, c \rangle P$ ”. The tree like graph illustrates Q and R co-operating with one another on no activities ($\langle \rangle$), effectively running in parallel. The components Q and R do however co-operate with component P on some activities (a , b and c).*

the visual navigator. Three particular factors were considered:

1. Since the PEPA plug-in was for the Eclipse platform, the technology used should be one that would integrate well into the IDE
2. Identification of current visualisation technologies and libraries that allow graphs to be created easily. This would allow more focus to be placed on obtaining all the PEPA data and implementing the navigator functionality, without the need to create additional graphing libraries.
3. The visual navigator required mouse events and the linking back to the PEPA API; these functions would have to be supported by the technology

In terms of languages to use, the Java language seemed the most obvious and sensible. Since the PEPA Plug-in API was written in Java, and the Eclipse IDE is also written in Java, using the same language would reduce the chances of compatibility problems and allow for better integration. Therefore, the technologies and libraries considered had to either be written in Java or, at the very least, integrate easily into Java.

The Workbench Visual Debugger was also written in Java. However, the tool did not use a pre-existing graph library, instead defining its own graph objects and intertwining with the Visual Debugger interface code. This custom graph library was slightly limited compared to other, dedicated graph library projects and was tailored towards the Visual Debugger’s visual representation of the PEPA model. In addition, the interface was created using the Java Swing graphics package, a package which does not integrate well into Eclipse.

2.2.2.1 The Standard Widget Toolkit (SWT)

The first condition when deciding on what technology to use was to use one that would integrate well into the Eclipse IDE. The first technology considered therefore was the Standard Widget Toolkit (SWT)[6]. SWT is a Java widget toolkit similar to the Java Swing and AWT libraries. Unlike these libraries however, SWT is not only faster than Swing or AWT, but is the widget toolkit used by the Eclipse platform to create the IDE interface. The use of SWT would then allow better integration into Eclipse.

Solely using SWT, the code used in the original workbench visual debugger would be ported over, with the PEPA methods updated to use the Eclipse Plug-ins API and use SWT instead of Swing to create the interface. This approach has one major draw back: as mentioned previously, the visual debugger defined its own graph objects, using the Java2D libraries, and drawing algorithms. Although this could easily be used to re-implement the visual navigator on a newer platform, it would not be taking advantage of any modern libraries available that were designed specifically for drawing graphs and drawing them well, failing to fulfil the second criterion.

2.2.2.2 The Eclipse Graphical Modelling Framework (GMF)

The second technology evaluated - The Eclipse Graphical Modelling Framework (GMF) - appeared ideal. GMF allows the easy creation of visual editors that are integrated right into the Eclipse platform. These editors produce very impressive looking graphs and implement much functionality (such as a magnifying glass) as default.

The process involved in the creation of a visual editor in GMF is illustrated in Figure 2.3. A representation of the model is described in an Eclipse Modelling Framework (EMF) file. This describes all the components of the graph that will be represented. The visual representation of these components is then described in a file, as well as the tools that will be used to create the objects. These three files are combined into a Map file, which says which components are visually represented and the corresponding creation tools. This Map along with a different file representing the model are used to create a generator file, which in turn creates a “.diagram” plug-in project containing the GEF (Graphical Editing Framework) code that is used to create the editor. The code in this project can be edited to allow different functionality (e.g. mouse events).

GMF initially appear the ideal solution, as the programs created integrated easily into Eclipse (fulfilling criterion 1) and contained libraries to facilitate in the creation of graphs (criterion 2). Unfortunately, problems were encountered when

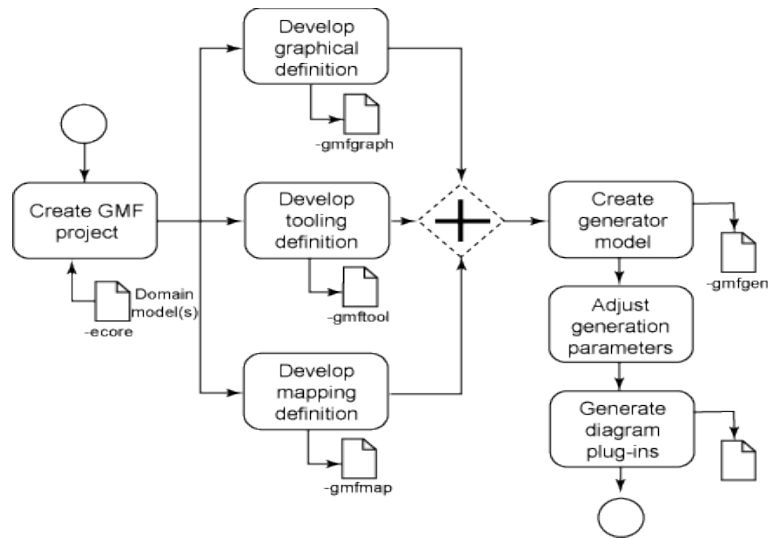


Figure 2.3: *The process involved in creating a visual editor in GMF. Picture from [3]*

learning about the technology. In attempting to do tutorials, it became apparent that major changes had been made from version 1.0 to version 2.0 (and then again from 2.0 to 2.2). Version 2.2 had been installed in the Eclipse program used for the project, which meant many changes needed to be made to get tutorials designed for versions 1.0 and 2.0, the majority, to properly compile. Once compiled, another problem arose in which all nodes were placed into the top left hand corner, one on top of the other, without the ability to be moved.

While investigating the cause of this problem, it became clear that the way the GMF created the visual representation of files may not be appropriate for the single-step navigator. The GMF editor stores the graphical information into two XML files (one the underlying model, the other the layout of the nodes on the canvas). This use of XML files meant that there was no direct or intuitive way to have the visual graph link to the PEPA API methods to allow interpretation of current states/next state. Without this ability to relate the on-screen representation to underlying Java code, it would be very difficult to implement the single-step navigator, failing the third criterion.

2.2.2.3 Zest: The Eclipse Visualization Toolkit

The third technology identified was the Eclipse Visualization Toolkit Zest[7]. Zest is a visualisation toolkit specifically aimed at the Eclipse platform, designed to integrate easily into programs using JFace or SWT. Originally released as a component of the Mylyn project, the project was ported over to the GEF project

and is now included as a standard component.

Zest uses the draw2d libraries to allow developers to create very attractive graphs with ease. The Zest library allows the creation of graphs, nodes and vertices through the use of *Graph*, *GraphNode* and *GraphConnection* objects. This then allows developers to treat the graphs as Java objects, without needing to worry about the maths required underneath.

In addition to allowing the creation of graphs, Zest also provides a series of layout algorithms for the automatic layout of graphs on the screen. These include *Tree Layout* and *Spring Loaded Layout*, the latter being the same algorithm used in the original workbench visual debugger.

The benefit of Zest over the other technologies is that, not only does it allow the creation of attractive graphs in a fairly straightforward manner, but is used as part of a Java application, which means mouse events and linking to the PEPA API methods is feasible.

2.2.2.4 The Final Decision

The final decision made was to use the SWT to create the visual navigator program and interface, with Zest being used to create the graphs representing the PEPA model. The SWT program would allow for good integration into the Eclipse IDE and would allow for all the functionality of the visual navigator to be implemented, while the use of the Zest libraries would allow for the creation and layout of the graphs to be made easier and simpler.

2.2.3 Interface Design

The final design decision relates to the visual navigator interface. The design of the interface is always subject to change, since limitations identified during the implementation stage may force redesigns, while new ideas may also be thought up at this time that had not previously been considered.

The initial design of the interface matches closely that of the Workbench visual debugger's interface. A window area holds the graphs representing the PEPA models, with forward/back and zoom buttons presented on the window. The current state of the machine is displayed on this window as well.

The visual navigator also offers a simulation mode, in addition to the single-step mode. The interface provides a clear distinction between these two different modes. In the Simulation Mode view, controls would be added to the window which are not present in Single-Step mode; the area being used by the graphs. To

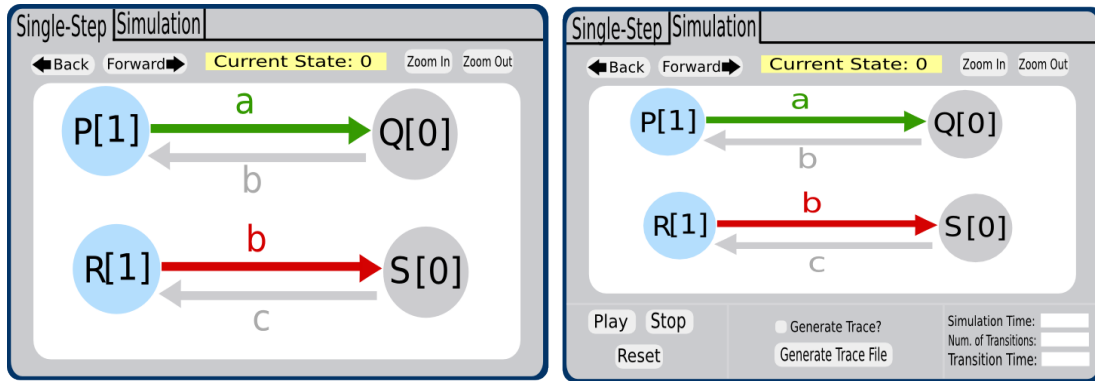


Figure 2.4: *The conceptual design for the new visual navigator. The left-hand side shows the visual navigator in Single-Step Mode, while the right-hand side illustrates the visual navigator in Simulation Mode.*

allow navigation between the two modes, two different methods were considered. The first idea has two separate tabs for the modes, where selecting the respective tab presents the window for that mode. This idea is illustrated in Figure 2.4. The second idea uses a menu option to navigate between the two modes, with the window view changing depending on which mode was selected. This latter choice would mean screen space would be saved but could hide the functionality from the user.

Although not represented in the diagram, two methods by which to view the System Equation were also considered: a tab option (which has the System Equation graph use the whole window area (i.e. no buttons or other widgets)) or a menu option that, when selected, brings up the graph of the System Equation in a pop-up window. The former option would keep everything in one window, allowing for a more consistent user experience, but the latter allows the user to view the System Equation in one window while performing single-step navigation or a simulation run in the main window.

3. Implementation

The implementation saw the creation of the Visual Navigator and comprised three separate sub-phases:

1. Development of a Java representation of a PEPA model suitable for visual modelling.
2. Using a supplied PEPA file, create a representation of it using the defined Java objects. This also required obtaining the state space information and instantiating the objects to reflect the current system state
3. Creation of the Visual Single-Step Navigator, including:
 - visualisation of the PEPA models (including aggregation and the System Equation)
 - implementation of the interface for single-step navigation (including highlight mode)
 - implementation of the simulation mode

3.1 Java Representation of the PEPA Model

When performing the initial research into which visualising technology to use, GMF was considered and some test examples created using it. This involved creating an EMF representation of the items to be represented in the graph, with nodes and connections being used to represent these items. For example, a “ProcessNode” object was created with attributes such as a name and aggregate value, which was represented on the graph as a node.

Even though GMF was not used for the actual implementation, the concept of having Java objects represent the model, objects that could then be mapped to a graph representation, was used. The use of Java objects instead of directly creating nodes and arcs allows for attributes to be defined for each of the objects that may or may not be represented visually (e.g. it may be useful to store the rate of an activity but not to show it visually); the actual visual representations may vary depending on the the type of object the graph item represents (e.g. for process nodes the name is required, for intermediate nodes the name is not displayed); it allows for a new graphing library to be used easily, simply by stating these objects should represent nodes and arcs in the new library.

The whole PEPA model is contained within a *PEPAModel* object. This model contains a list of *Rate* objects, a list of *PEPAComponent* objects and a *SystemEquation* object.

3.1.1 Rate

This first object is an example of an object which, for this implementation, was created to ensure that all data about the PEPA model was stored and is not required to be represented visually in any form. A *Rate* object is either a user defined rate (e.g. “ $r = 2$ ”) or a value used in an activity (e.g. “0.2”, “1+2”, “inf”). Each *Rate* object has a name and value attached to it, with the name being either the name used in the user defined rate or the value for non-user defined rate.

The *PEPAModel* contains a list of all the rates used within the PEPA model.

3.1.2 PEPAComponent

A *PEPAComponent* object represents a component of the PEPA model. A component of the model will contain some number of processes, activities and, possibly, intermediate states. Each *PEPAComponent* corresponds to a distinct graph representing the PEPA model visually. Therefore, the object contains the objects that will later be mapped to nodes and connections.

- *ProcessNodeVis*

This object corresponds to a Process state in the PEPA file (e.g. “PC10”) and extends a generic *Node* class. The name *ProcessNodeVis* was used due to another item in the PEPA API having the name “ProcessNode”, the “Vis” identifying this object as the process node object used for the visualisation. Since this object corresponds to a process state, the object has a name attribute, a boolean attribute stating whether the process is a current state for the component and an aggregate variable containing the current aggregate value of this process node. These variables are defined in the *Node* class.

- *IntermediateNode*

An *IntermediateNode* also extends the *Node* class and has the exact same attributes as a *ProcessNodeVis*. The difference is that *IntermediateNode* objects represent the intermediate states of the component (as mentioned in previous sections) and will be treated slightly differently when visualising.

- *Activity*

This object corresponds to an activity in the PEPA model. An activity has a name assigned to it as well as a *Rate* object. The original PEPA file describes a series of transitions for each process (e.g. “ $P = (a, r).Q$ ”). Since the activity objects are to represent connections between processes and/or intermediate nodes, the *Activity* object contains a destination and origin variable containing the *Node* object corresponding to these. Using the previous example, the transition would be reflected by an *Activity* object named “a”, with the origin indicated by a *ProcessNodeVis* named “P” and the destination indicated by one named “Q”.

In addition to these attributes, the *Activity* object contains an *ActivityState* object (an enumeration class, containing “NEXT”, “BLOCKED” and “BLANK”) representing the state of the activity, in addition to a unique id to allow identification of each activity.

3.1.3 SystemEquation

The final object represents the System Equation of the PEPA model. The System Equation is composed of components and activities the components cooperate on. This is represented in the *SystemEquation* object as two lists of objects: *SEComponent* and *SECooperationAction*. For this sub-section, the System Equation “ $(Q \langle \rangle R) \langle a, b, c \rangle P$ ” will be used as an example.

- *SEComponent*

A component in the System Equation represents a component of the PEPA model. In the example, “P”, “Q” and “R” are the components. An *SEComponent* is created to represent each of these, which contains a name attribute, an aggregate value (the System Equation states how many of each components are represented in the PEPA model) and an ID to a *SECooperationAction* object.

This final attribute identifies which list of co-operation activities the component “directly” co-operates on. In the example, the components “R” and “Q” directly co-operate on the “ $\langle \rangle$ ” co-operation, but also indirectly on the “ $\langle a, b, c \rangle$ ” co-operation. The “P” component co-operates directly on this latter co-operation. The notion of “direct” and “indirect” co-operation is intended to reflect both the manner in which the System Equation should be interpreted and the way the eventual graphical representation of the equation should look.

- *SECooperationAction*

This object represents a co-operation of activities in the System equation. The example above contains two of these co-operation activities: “<>” and “< a, b, c >”. To fully represent a co-operation activity, each *SECooperationAction* object contains a list of Strings containing the names of the activities involved in the co-operation (possibly empty if the co-operation activity represents components that run in parallel), as well as the unique ID mentioned previously.

In addition, the ID of the parent *SECooperationAction* is stored. Only components co-operate with one another over a list of activities, though a group of components may co-operate with another group over the same set of activities. In the example, “R” and “Q” both co-operate with “P”, however “R” and “Q” co-operate with one another over a different set of activities. This arrangement is represented by stating that the components “R” and “Q” co-operate over “<>”, while “P” co-operates with “<>” (meaning the “sub-tree” or “sub-components” of the co-operation) over “< a, b, c >”.

The top-most co-operation activity, in this form, will not have a parent co-operation activity and so will have “-1” as the parent ID. This value was chosen because assigning the root co-operation activity an ID of “0” seemed most logical and, since all other IDs will be greater than this ID, “-1” would remain unique.

3.2 Obtaining the Data about PEPA Model

Using the representation of a PEPA model defined in the previous section, the functionality to create a *PEPAModel* object of a user selected PEPA file was implemented. The file selected is assumed to be a syntactically correct file, as the visual navigator will check the file and stop if it is invalid PEPA code. This involved parsing the input file - using an AST representation - and creating the necessary objects.

Once the *PEPAModel* representation has been created, the state space has to be derived (i.e. find all possible configurations for the system and store the information in a variable). With the state space information, the state of the various processes and activities can be instantiated for a given state index (e.g. for state i , which processes are the current for each component, which activities can fire, what are the aggregate vales etc.).

The *PEPAData* object provides this functionality. The class constructor takes a PEPA file as a parameter and contains a *PEPAModel* object, which is instantiated using the provided file. The class provides various methods to determine state

information and instantiates the process/intermediate nodes and activities based upon this information. In addition, methods are provided that are used by the visual navigator to determine what the current state is, what state to transition to and update the components based upon the new state.

3.2.1 Instantiating the PEPAModel Object

Creation and instantiation of the *PEPAModel* object occurs in the constructor of *PEPAData*. The constructor takes, as a String, the location of the PEPA file to use and converts it into an AST, by creating a *ModelNode* object, in the form described in section 2.1.1. This AST is then parsed to get the Rates, Processes, Activities, Intermediate State (if any) and System Equation.

The user defined rates are first obtained from the AST tree. A method is called to extract all the children of the “RATES” node from the *ModelNode* object. For each rate defined in the AST, a corresponding *Rate* object is created, using the child name and value, and added to a list of all the rates.

Once the user defined rates have been recorded, the next method called iterates through all the children of the “PROCESSES” node of the AST, creating *ProcessNodeVis* objects for each. The child’s child “name” is used for the objects name, with an aggregate value of 0 and a current state status of *false* assigned initially. Each of the *ProcessNodeVis* objects are added to a list containing all the processes. The children of the “PROCESSES” node have the corresponding transition as another child, the information for which is extracted in a separate method. This decision requires the tree to be iterated through the children of “PROCESSES” twice. Although this does have some minor performance problems, the decision allowed for the creation of code that was easier to understand, follow and, most importantly, maintain.

The transition data of the different processes is then examined. Every process in the PEPA model has an associated “equation” stating the one or more possible transitions from this process state to another. These transitions comprise one or more activity sequences followed by a destination process. The destination can be either a different process or the same (a loop). Each transition is inspected and represented in the *PEPAModel* with *Activity* objects, and possibly *IntermediateNode* objects, being created. This is one of the more complex parts in the creation of the *PEPAModel* object, involving two stages and several method calls:

1. *Obtain each transition*

The first step is to obtain all of the transitions defined in the PEPA Model. The algorithm used to perform this operation is presented in Algorithm 1. Each of the process definition children of “PROCESSES” is inspected to

obtain the name of the process (the origin of the transition) and the process equation. This equation is then inspected to discover how many transitions are defined. For each transition, a vector of String objects is created: the first and last entries are the process origin and destination of the transition respectively; the Strings in the middle being the activities that must fire in the transition. This vector is then passed along to the *getTransition* method, defined in Algorithm 2.

A single transition is represented within a *PrefixNode* object. When a *ChoiceNode* is encountered, this means there are two or more transitions. A method is called on the left and right hand side of this node to obtain a list of all the *PrefixNode* objects contained in the branches. A method is then called on each *PrefixNode* to obtain the vector of Strings, which is then passed to the *getTransition* method.

The method to create the vector of Strings representing the transition takes a *PrefixNode*, extracts each activity in the sequence (recreating the String representation), adds the String representation in the order they must fire, then gets the name of the target process and appends this onto the end. The origin process is appended to the front of the list prior to being sent to the *getTransition* method.

2. *Given the Vector of a transition, create the necessary Activity and IntermediateNode objects to represent it*

Once a vector representing a transition has been created, the vector is examined and the appropriate *Activity* and *IntermediateNode* objects are created to have the transition represented in *PEPAModel*. Three conditions are checked, with different evaluations for each:

- (a) *Less than 3 items in the vector*

A transition is represented by a start process, an end process and at least one activity in between, giving a vector of length three in the simplest case. If it is less, an error has occurred.

- (b) *Exactly 3 items in the vector*

If length is three exactly, this is the simple situation where a single activity causes a transition from one process state to the next. A new *Activity* object is created as follows:

- The name for the new activity comes from the String representing the activity (the second String in the list).
- The *Rate* for the new activity comes from using the name of the rate seen in the activity String and using the corresponding *Rate* object with that name.

- The origin node is the *ProcessNodeVis* with the same name as the first String in the vector
- The destination node is the *ProcessNodeVis* with the same name as the last String in the vector

This *Activity* object is then added to the list of activities and the method is complete.

(c) *More than 3 items in the vector*

This is the more complex situation. When there are more than three items in the vector, the first and last entry correspond to the origin and target processes respectively, but now more than one activity must fire before the transition is completed. This means that at least one *IntermediateNode* must be created. The first *Activity* object is created in a similar fashion as before, except this time the destination node is an *IntermediateNode* node. The name of this node is constructed by taking the remaining entries of the vector (e.g. if i th entry is the activity, the entries $i+1$ to the last entry), appending them into a String with a full stop between each. This representation is used as it matches the way intermediate states in the components are represented in the state space (meaning it can be easily mapped to a state in the state space). If an *IntermediateNode* by this name is not already in the list of all intermediate nodes, a new one is created and added to the list.

Algorithm 2 demonstrates the steps involved in creating all the necessary *IntermediateNode* and *Activity* objects. Essentially, the activities in the vector are iterated through, with the previous *IntermediateNode* being the origin node for the next *Activity*, a new *IntermediateNode* being the destination. Once the last activity in the vector has been inspected, the *ProcessNodeVis* corresponding to the last String in the vector is used as the destination and the last *Activity* object is created and added to the list. The method is then complete.

Once the information about the rates, processes and transitions in the PEPA file has been used to create the *Rate*, *ProcessNodeVis*, *Activity* and *IntermediateNode* objects needed to describe the PEPA model, the final piece of information to be extracted is the System Equation. The algorithm used to obtain the *SystemEquation* object for the PEPA file is given in Algorithm 3.

The *ModelNode* containing the AST of the PEPA file is used once more to extract the “System Equation” child of the tree. This node contains the System Equation and is almost always a *CooperationNode*. This type of node has three children: a left, right and actions child. In the most simple situation, where the System Equation contains just two components co-operating over one set of actions, the

left and right children will point to the respective components. However, care needs to be taken in two additional cases, the first when the component has an aggregation value attached - this value must be extracted and stored (when no aggregation value is assigned, an aggregate value of “1” can be used). The second situation occurs when there are more than two components in the system equation. Here, the left and/or right child could be another *CooperationNode*, in which case this branch has to be further investigated to extract the full equation.

The process of creating the *SystemEquation* object works as follows¹:

1. a list of *SECooperationAction* and *SEComponent* objects are created, to store the co-operation actions and components respectively
2. a variable, *id*, is set to 0, while another variable, *parentID* is set to -1.
3. The *ChoiceNode* is extracted and a do...while loop is entered
 - (a) The actions child is inspected. The name of each action is extracted and added to a list. This list, along with the current *id* and *parentID* values, are used to create a new *SECooperationAction* object. This is then added to the list
 - (b) Next the left child is examined, with a series of if-elses used to determine the child’s type. If the child is a *CooperationNode*, a flag is set to indicate this left branch needs to be further expanded. Otherwise, the node represents a component and a new *SEComponent* object is created using the components name, the aggregate value being either the one given or “1”, and the current value of *id*. In this case, the left branch flag is set to false
 - (c) The right child is then examined, the steps required being identical to the left node except that a separate right-branch flag set true/false.
 - (d) The flags are then checked: if the right flag is set true, the right *CooperationNode* will be examined in the next iteration; else, if the left flag is true, then that branch will be examined
 - (e) The *parentID* variable is given the value of *id*, the value of *id* is incremented
 - (f) The do...while checks to see if either flag is true
4. Once iterated through, a new *SystemEquation* object is created using the two lists of *SECooperationAction* and *SEComponent* objects.

¹An iterative approach was chosen over a recursive one due to data flow requirements; the correct values for the IDs and parent IDs had to be guaranteed and it was felt a recursive method could not guarantee this.

As stated in the previous section, each co-operation activity set has a unique ID, which is used to identify which components co-operate over the actions. A “parent ID” allows for identification of when the sub-tree of a co-operation activity co-operates with another component. The method used to create and assign the ID’s means that each co-operation set has a unique ID and that each component is assigned the correct ID, while each component is assigned the correct parentID. There is one problem in the situation where both left and right branch are *CooperationNode*’s, which is discussed later in the *Evaluation* chapter.

After all the information has been recorded, the *PEPAModel* is created and instantiated. The *Rate* list and *SystemEquation* objects created so far are added directly to the *PEPAModel*. Finally, a list of *PEPAComponent* objects is created and included in the *PEPAModel*.

To create the list of *PEPAComponent* objects, three lists of *Activity*, *ProcessNodeVis* and *IntermediateNode* objects are required. These lists contain the objects that belong to each of the components. However, to determine what these components are, and how many of them there are, the *SystemEquation* object’s *SEComponent* object list is used. The size of this list determines the number of *PEPAComponent* objects to be created. The steps involved in creating each *PEPAComponent* are as follows:

- For each *SEComponent*
 1. Get the component name
 2. Find the *ProcessNodeVis* with this same name, add to a temporary list
 3. Find all *Activity* objects with this process as either destination or origin, if not already in the activities list, add
 4. When the process was the origin, find all the *ProcessNodeVis* and *IntermediateNodes* that were the destination, add to the lists if not already there and look through the activities where these nodes were either destination or origin (adding if new activities/nodes found, breaking out when no new items found)
 5. Do the same check but for the *ProcessNodeVis* and *IntermediateNodes* where the origin and the first process was the target.
 6. Once all the information for this component is acquired, create a new *PEPAComponent* object using the temporary lists of *Activity*, *ProcessNodeVis* and *IntermediateNode* objects. Add this *PEPAComponent* to the list.

With the components created, the *PEPAModel* object is fully instantiated and represents the PEPA model defined in the PEPA file. The next step is to calculate

the state space and set the state of the various processes and activities for the initial state.

3.2.2 State Space and State Information

To obtain the state space for the model, a method provided by the PEPA Eclipse Plug-in API takes the *ModelNode* object and returns an *IStateSpace* object. This latter object encapsulates all the state space information, including all possible states of the PEPA model, probability for being in each state, which transitions are possible from one state to another and activities that can fire in that state. This object is used extensively to obtain and instantiate state information. For the constructor, the current state is set to the initial state index i.e. “0”.

The first step involves deciding which activities can fire in the initial state. A method is called that takes the initial state index and uses it to extract the state information for the initial state. All activities in the PEPA model have their state set initially to blank (required for resetting activity state). The state is then examined to see the names of activities that can fire. Each name is then sent to a separate method which examines the list of activities and, for those with the same name, sets the activity state to “NEXT”. This initial assignment can label some arcs as being fired that cannot (due to processes having an activity with the name but not being a current state). This is dealt with later; at this point, activities are either next or blank.

Once the activities that can fire have been established, the next method called takes the initial state and determines the current state for each component. The current state for a component may be one process or intermediate state, or more than one if aggregation is used. For example, “P”, “(a, r).Q” or “P[1], (a,b).Q[1]”. The state space object is again used to obtain the component information. This component information provides the name of the current process or intermediate state(s) for each component, along with aggregation values if aggregation was used with this component. Two lists are created to store the information: one for the name of the current state, one for the associated aggregate value. The method iterates through each component from the state space object and performs the following checks and actions:

- *If the label for the component does not have a “[”*

No aggregation was used for the component in the System Equation, so the label describes just one process/intermediate state. Add the name to the name list and an aggregate value of “1” to the other list.

- *If the label for the component does have a “[”*

Aggregation has been used. This means that one process/intermediate state may be the current state or a number of them (note, however, the initial state will always be just one process state as the current state). In order to extract all the necessary information, the label has all occurrences of “[”, “]” and “,” removed and split into an array, such that the even index gives the name of the state and the odd index the aggregate value for the previous index. For example, “P[1], (a,b).Q[1]” \implies [“P”, “1”, “(a,b).Q”, “1”]. This array is then iterated through, with each name and aggregate value being added to the respective list at the same index.

Now that the current state for each component is known, as well as the activities which can fire in this initial state, the final method takes this information and sets the state and aggregate value for each *ProcessNodeVis* and *IntermediateNode* object. An assumption is made at this point that all processes and intermediate states have a unique name, an assumption that does not conflict with the PEPA semantics. Each *ProcessNodeVis* and *IntermediateNode* object in the PEPA model initially has the current state status set to false. The list of names that are current states are then iterated through. When the corresponding *ProcessNodeVis* and *IntermediateNode* object is found, the object’s current state status is then set to true and the aggregate value of the object is set to the value in the second list.

Once all these processes and intermediates have had their current state status set, the final step is to finalise the activity states. Previously, those activities that had the same name as an activity that can fire were set to “NEXT”, the rest being set to “BLANK”. Now, each activity is checked again:

- if the origin is not a current state and the activity has been set to “NEXT”, the state is reset to “BLANK”. The activity may have the same name as an activity that can fire, but since the origin is not a current state this specific activity cannot fire in the current system state
- if the origin is current and the activity has been set to “BLANK”, then this is an activity that cannot fire in the current system state. However, since the origin is a current state, the reason the activity cannot fire is because it is part of a co-operation and the other component(s) it cooperates with is not in the correct state. Therefore, the activity’s state is changed from “BLANK” to “BLOCKED”

Once all these methods have been invoked, a representation of the PEPA model defined in the file has been created and stored as a *PEPAModel* object and configured to reflect the system in the initial state.

3.2.3 Changing State and Updating

The implementation thus far allows for a PEPA model to be created based upon a PEPA file and instantiated to reflect the initial state (with this initial index value stored in a variable). For the Visual Navigator, it must be possible to not only change the system state but to also update the model for this new state.

To perform a change of state, a “change state” method was implemented. This method, when provided with a state index, checks to see if it is a valid index value for the current PEPA model and, if it is, sets the current state index variable to this new index and invokes the methods, described in section 3.2.2, to update the PEPA model for this new state.

The methods implemented to instantiate the model for the initial state are re-used when changing state, and as such do not need to be discussed again here. The real challenge for this part was determining what the state index of the next state should be. In terms of the Visual Navigator, the only information that will be available will be knowledge of which activity arc the user clicked on and what the current state is. In order to determine what the next state index should be, a final method is provided in *PEPAData* which, given the following:

- *the state index*
- *the activity name*
- *the origin node’s name*
- *the destination node’s name*

will return the index of the state the system enters when this particular activity is fired in the system state given by the state index (or “-1” if the actions cannot be fired in this state).

As a first step, the method uses the name of the origin, in conjunction with the state index, and the state space object to determine which component the activity arc belongs to. Once the component has been found the next step involves checking to see if the component label contains a “[”:

- If the label does not, then aggregation has not been used for this component and so this does not need to be taken into account. Every activity that can fire in the index state is checked to see if the activity has the same name as that provided to the method. If it does, the index of the state entered taking this activity is noted. Then, the label of the component in this tentative new state is checked; if the name is the same as the destination, this is the transition that should take place (since the component’s current state changes from the origin to the destination). This new state index is returned.

- If the label does, then aggregation has been used, meaning some additional checks must be carried out. To make sure the right state to transition to has been selected, the change in aggregation value is checked in addition to the steps carried out before. The steps involved are as follows:
 1. obtain the aggregate value for the origin in the index state. Then check to see if the destination is also in index state i.e. a current state for the component in the index state as well (since the state label may be of the form “P[1], Q[2]”, where P may be origin and Q destination). If the destination is not in the index label, set the aggregation value to zero, else, use the aggregation value stated.
 2. check to see if an activity with same name can fire in the index state
 3. if one can, check to see if the state transitioned to contains the destination as a current state for the component (same as before) and store its aggregation value. Check to see if the origin is there too (either use aggregation value found or zero)
 4. if the potential next state does contain the destination, the final check is to decide whether this is the correct transition to perform by inspecting the aggregation values:
 - if the origin and the destination have the same name, this is a loop from the current state back to itself. The aggregation value of the origin in the index and potential next state should not change, similarly for the destination aggregation value.
 - if origin and destination are different, the origin aggregation value should be one less in the new state, while the destination aggregation should be one more
 5. If this final aggregation value check passes, the next state has been found. This state index is returned as the target system state.

At this point, all the methods and objects are in place to create a Java representation of a PEPA model. The model can be instantiated to represent any state in the system state space, as well as updated dynamically to reflect a different state that has been “transitioned” to. The next step involves creation of the Visual Navigator. This stage involves creation of the interface; the visual representation of the PEPA data information; the implementation of the Single-Step navigation and Simulation modes.

3.3 Visual Navigator

The final phase of the implementation stage saw the creation of the visual navigator program itself. This final phase consisted of three sub-phases: creating the graphical representation of the PEPA model; creation of the Single-Step navigator interface; implementation of the Simulation Mode.

3.3.1 Graph Representation of PEPA Model

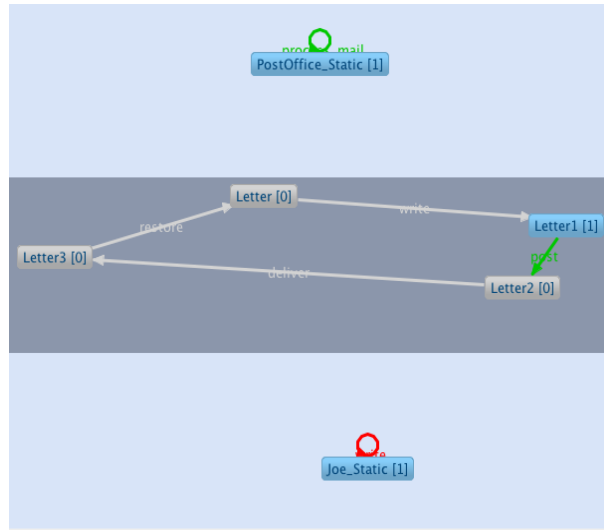
The Java objects created to represent the PEPA model, along with the class which creates a Java representation of a provided PEPA file, are utilized in this first sub-phase. Initially, two separate programs were created: one which created a visual representation of the PEPA model; another creating the graphical version of the System Equation.

3.3.1.1 PEPA Components

The creation of graphs representing the PEPA components was fairly straightforward by design, the Java objects being defined in such a fashion that allowed a direct correlation to the graphical representation. The instantiated *PEPAModel* object is used to create the visual representation through the following steps:

- For each *PEPAComponent* object
 1. Create a new graph for the *PEPAComponent*
 2. For each *ProcessNodeVis* in the component, add a new node to the graph, giving it a name composed of the objects name and aggregate value
 3. For each *IntermediateNode*, create a corresponding node, using the aggregate value as the graph node's name
 4. For all the *Activity* objects, a new graph connection is created with the name the object had. The origin and destination are set by checking to see which *ProcessNodeVis* and/or *IntermediateNode* were used and using their corresponding graph node.
- Once the graphs have been created, a “re-draw” method is called, which checks the state of each objects and sets the colour of the corresponding graph object accordingly

The resulting visual representation is presented in Figure 3.1 (this is the visual representation of the *Letter.pepa* file in state index 1). The figure also presents a

Figure 3.1: *The Visual PEPA Model*

design decision made during the implementation stage: alternating background colours for each PEPA component graph. When creating multiple graphs, the Zest library assigns each graph its own, unique window space. With a simple white background for each graph, the distinction between components is not as apparent. Alternating between a blue and blue-grey colour for the background allows for a greater distinction between both the graphs and the components they represent; further making concrete that, although connected, the components are separate elements of one model.

For automatic layout, the built-in “spring-layout” algorithm is used. This algorithm is the same one used by the original workbench virtual debugger. Once the automatic layout has been performed, the nodes can be moved around the graph window by the user

3.3.1.2 System Equation

The second program takes the same instantiated *PEPAModel* used in the previous program and extracts the *SystemEquation* object for the purposes of drawing the graph:

- Each *SEComponent* has a graph node created for it. The node is given the same name as the component object for the label and the colour is set to orange.
- Each *SECooperationAction* then has a graph node created for it. The name for these nodes is constructed by placing the name of each action in a list

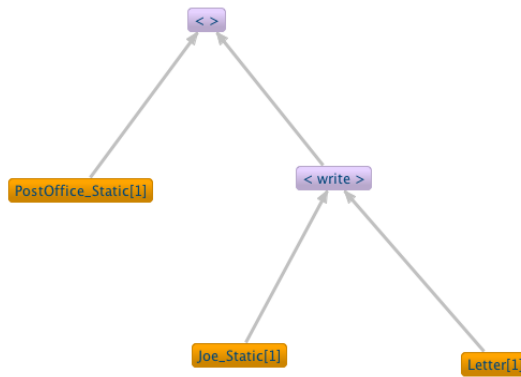


Figure 3.2: *The Visual System Equation*

surrounded by “<>”. A purple colour is then assigned to the nodes

- Finally, the connections are created. The nodes representing the cooperations are assigned connections first: each objects “parentID” value is checked and - if it is not -1 - the corresponding co-operation node is found and a connection created between the two. The component nodes are then checked, with the “id” value used to determine which co-operation node a connection should be drawn to.

The graphical representation of the System Equation for the *Letter.pepa* file is presented in Figure. 3.2.

3.3.2 Single-Step Mode

Once the representations produced by the programs created in the previous sub-phase were successfully tested, work began on the visual navigator proper. The program produced in this sub-phase integrated the two previous programs into one interface, as well as implementing single-step navigation and highlight mode.

3.3.2.1 The Interface

Once a PEPA file has been selected, the visual representation of the file is created underneath and the user is presented with the screen shown in Figure 3.3.

The top row of the interface contains backwards and forwards buttons which, once the user has navigated to different states, can be used to return to a previously visited state. The top row also contains some text stating what state index the current view is representing (state 0, or the initial state, in the figure).

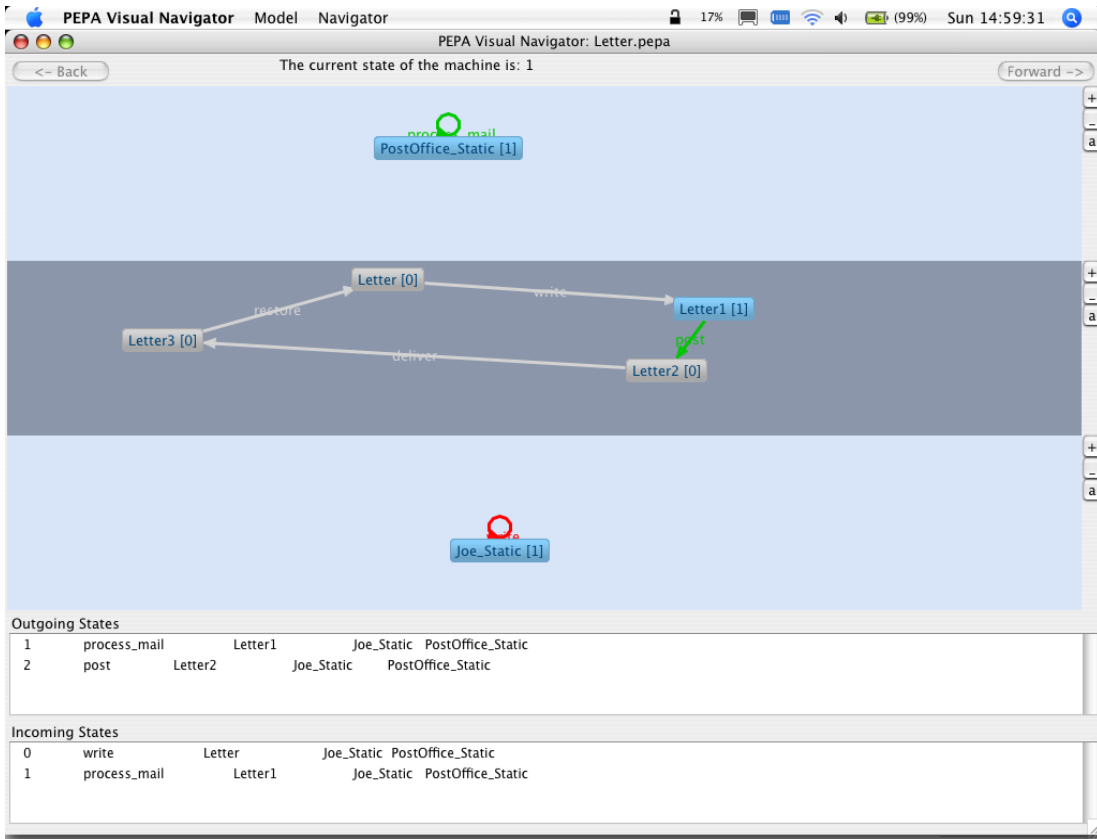
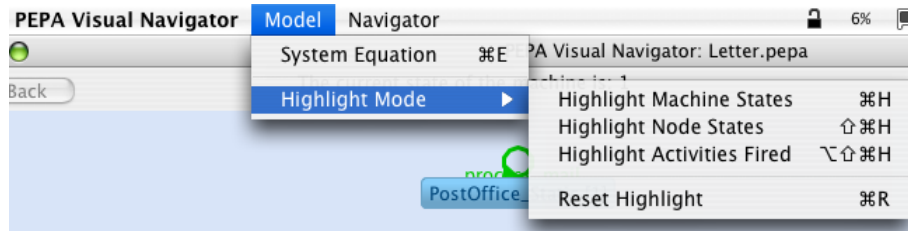
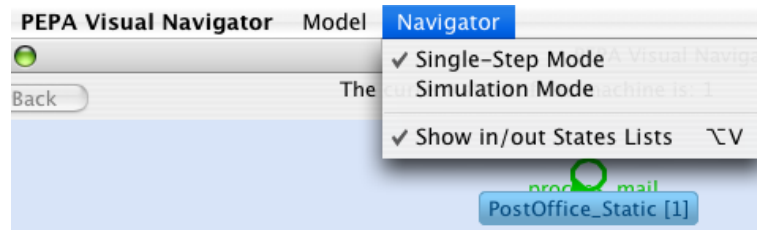


Figure 3.3: *The Visual Navigator in Single-Step mode (Letter.pepa)*

The middle portion of the interface is allocated to the graphs. The area designated for the graphs is fixed (in relation to the window size, explained later), with the size of each graph being calculated by taking the number of graphs being produced and dividing the vertical space equally between each. Beside each graph are a set of three buttons, which allows the user to customise the view of the graphs: the top “+” button maximises the graphs (makes that graph cover the whole graph area and hiding any others); the middle “-” button minimises the selected graph (hiding that graph and dividing the graph area between the remaining, visible graphs); the bottom “a” button toggles the visibility of the activity arcs (allowing the user to view just the possible states of that component). The three buttons are all toggle buttons, i.e. when pressed they stay depressed and shaded differently until pressed again. The interface has been designed such that, if a maximise button is currently on and another maximise or minimise button is selected, the first maximise button is turned off and the new instruction executed as if no maximise or minimise buttons had been initially selected; similarly, when a minimise button is depressed and a maximise button is then pressed. When a minimise button has been pressed and another is then pressed, both graphs are minimised. These maximise/minimise decisions were made as it

Figure 3.4: *The Visual Navigator in Model Menu*Figure 3.5: *The Visual Navigator in Navigator Menu*

was simpler to program while still being intuitive for an end-user to use.

The final portion of the interface contains two lists, displaying possible incoming and outgoing states from the current state. The inclusion of these lists was due to a slight limitation in the current version of Zest (explained later in the section). Since these lists were not intended as a permanent inclusion, a menu option has been included to toggle the visibility of the list. When selected to not be shown, the extra space is allocated to the graph area, with the size of each graph recalculated to include the new area.

The interface also includes two menus: *Model* and *Navigator*. The *Model* menu, shown in Figure 3.4, contains two items relating to the PEPA model. The first is the System Equation which, when selected, produces a pop-up containing the visual graph of the System Equation. The second item contains a sub-menu to the various highlight modes (discussed later). The *Navigator* menu contains operations that may be performed on the Visual Navigator, including selecting between Single-Step and Simulation mode, as well as toggling the visibility of the state lists. This second menu is shown in Figure 3.5.

As a final note, the layout and resize code has been custom made for the Visual Navigator interface. The available layout algorithms did not allow for the elegant placement of widgets, instead placing each item in a row of its own (horizontal layout) or in a column of its own (vertical layout). Another problem arose from including the lists; when a large number of possible states could be next, the layout algorithms would assign a large amount of space for the lists in order to display all the next states. To create the desired interface layout, the positioning

of each widget/graph was carefully calculated and placed. When the user re-sizes the window, the placement of each is recalculated and positioned to conform to the new window size. For example, when the height of the window is increased, the top row and bottom lists are kept at the same height, with the additional height being given to the graph area, resulting in each graph being taller (a design decision taken because the main focus for the interface should be the graphs).

3.3.2.2 Single-Step Navigation

The original concept for single step navigation involved the user clicking on an activity arc that was coloured green² and the visuals being updated to reflect the next state. A slight redesign was made to have this action be a double-click, as it was felt a double-click was a more intentional action to perform, meaning the likelihood of accidentally firing an activity was reduced.

When a user double clicks on an activity arc, the following actions are performed internally:

- A check is made to see if the arc is coloured green. If it is not, the activity cannot fire and so no further action needs to be taken
- If the activity is green, the associated *Activity* object is obtained, with the activity name, origin name, destination name and current state index sent to the method described previously.
- The state index returned by the method is then sent to an *update current state* method, which changes the current state to the new one and updates the states of all processes, intermediates and activities for the new state
- The graph re-draw method is invoked, recolouring each of the nodes and arcs to depict the new system state.

Although the double click method works accurately and as expected, a constraint with the current version of Zest became apparent during the implementation stage. Currently, Zest can only draw straight connection arcs between nodes, the functionality to create curved connections being temporarily removed. This removal was due to the current migration of Zest, to allow for an easier transition period, and will be reintroduced in a later version. This causes a problem when more than one activity connects two nodes, as the straight lines overlap one another. This means that double clicking on the last arc drawn is easy, but on ones drawn prior to this, very difficult. As a temporary solution, the outgoing/incoming state lists were included. Displaying all the possible states that can

²The colour of the activity arc identifying the value of the corresponding *Activity* object's state attribute; the colour green meaning the state attribute value is "NEXT" (i.e. that it can fire)

fire from this current state (or may have fired to lead into this state), the user can perform single-step navigation by clicking on one of the activities in the list. From the user's perspective, this performs the same action as double-clicking an arc (although is slightly simpler to deal with internally, as the next state index is instantly known). The use of menus to show the possible next/previous states, and allowing user to select the next state from the list, was inspired by the text based Single-Step Navigator.

3.3.2.3 Highlight Mode

The final component of the Single-Step mode is the Highlight mode. When deciding how to visually highlight the graphs during the design phase, three ideas were considered. Each idea had their merits on paper, so it was decided to implement all three versions of the highlight mode; this made it easier to evaluate the pros and cons of each design idea. This version of the Visual Navigator includes all three versions, allowing the user to choose the one they feel is most useful. The three modes are as follows:

- *Highlight Machine States*

The first highlight mode allows for the instant identification of when the system re-enters a system state it has been in before. A list of the index values of system states is kept for this highlight mode. When a system state is entered, the current state index is checked against the list, being added if it was not already there. If this highlight mode is on, the list of previously visited states is checked: if the current state has been visited previously, all nodes and arcs are coloured in a darker shade of their respective state colour (e.g. dark blue for current state nodes, dark red for blocked activities); if the state has not been visited previously, the state index is added to the list as usual and the graph coloured normally. Figure. 3.6(a) illustrates this highlight mode.

- *Highlight Node States*

The second highlight mode highlights nodes a darker shade of their current state colour (dark grey or dark blue) if this node has had the value previously. For example, if node "P" has had the value "0" before, it will be coloured dark grey, while if "Q" has not been "5" before, it will be coloured the normal blue shade this time, but if the system state is changed to one where "Q" again has the value "5", it will be dark blue. This is shown in Figure. 3.6(b).

Similar to the previous highlight mode, a list of Strings created by appending the node's name with it's aggregation value is maintained. When a new

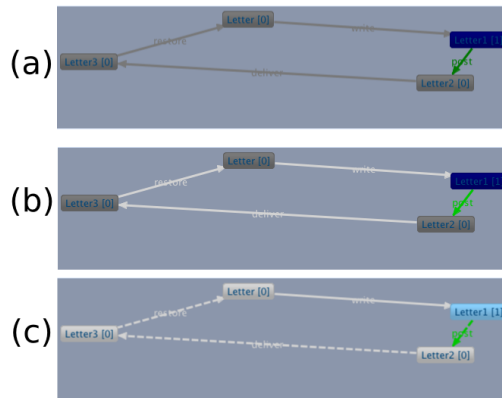


Figure 3.6: *The Visual Navigator's various Highlight Modes (a) Machine Highlight Mode (b) Node Highlight Mode (c) Activity Highlight Mode*

system state is entered, each node is checked against the list, being added if it is not there with the current aggregation value already.

- *Highlight Activity States*

The final highlight mode highlights activities that have been fired previously. When entered, all activities that have not fired are represented by a dashed connection. When an activity is fired, or has been fired, the activity arc turns solid. This mode allows the user to see when no activities have been fired (all connections are dashed lines), when all activities have fired (all connections are solid) and every state in between.

Identifying which activities have been fired requires two steps. First the name of the activity that has been fired (either by double clicking or selecting from the list) is obtained. Then, each arc with that name is checked: if the arc is green (meaning it could fire in this state) and is currently a dashed line (meaning it has not been fired) then the connection is set to solid and a unique ID, assigned to the activity previously, is added to a list. This latter step allows the user to toggle the highlight mode on and off and still have an accurate view of the fired activities (the ID list being used to decide which arcs should be dashed and which solid).

Figure. 3.6(c) is an example of this final highlight mode in action.

When the visual navigator is first started, all the information required by the highlight modes is recorded. This means that the user has an accurate representation of the history from the first time they enter the highlight mode. A reset option is also available, allowing the users to clear the history and start out fresh.

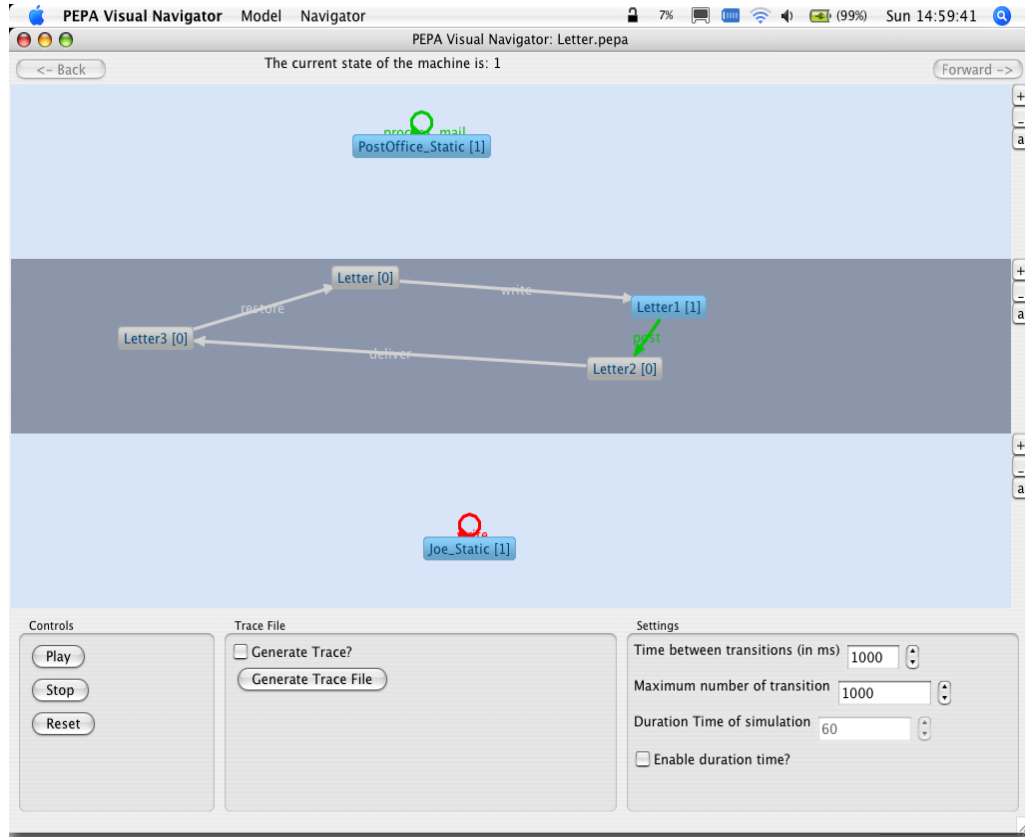


Figure 3.7: *The Visual Navigator in Simulation mode*

3.3.3 Simulation Mode

The second mode available to users of the Visual Navigator is the Simulation mode and was the last component of the system to be implemented. The Simulation mode - similar in functionality to the feature available in the Workbench Visual Debugger - allows the user to have Single-Step navigation performed automatically, with the computer deciding which transition to fire next. Figure 3.7 shows the Simulation interface. With each transition fired, the PEPA models are updated to reflect both the new state of the system and also the history of nodes visited and activities fired. The mode also provides the user with the ability to customise the simulation run, as well as generate a trace file describing the simulation run.

3.3.3.1 Entering Simulation Mode

In the original Workbench Visual Debugger, the Simulation mode buttons were integrated as part of one window, meaning that window space was assigned to

simulation buttons even if the user was not intending to use the feature. In order to avoid this, as well as create a greater distinction between the Single-Step and Simulation modes, the Visual Navigator keeps these modes separate. In order to change between modes, the user must select one from the *Navigator* menu.

When entering Simulation mode, the lower portion of the screen, assigned to the next state lists in Single-Step mode, displays the different controls available to control the simulation run. Therefore, the general interface has a similar appearance in either mode - creating a sense of cohesion - while at the same time allowing for a clear visual recognition of which mode the user is currently in.

An important consideration had to be made when deciding what should happen when the user transitioned between Single-Step and Simulation mode. Since the Visual Navigator provided many options to allow the user to customise the view of the graphs, the following situations had to be considered and a solution found:

- *The user selects the Simulation mode while having a Maximise or several Minimise/Hide Activity Arcs buttons depressed*

In this first situation, the user has customised the view of the graphs into the manner felt best. Therefore, this view is respected when changing into Simulation mode. Visually, the layout and presentation of the graphs in this respect look the same when the user goes from Single-Step \rightarrow Simulation mode, and vice-versa.

The buttons are also selectable while in Simulation mode, so the user may customise the graph views like in Single-Step mode.

- *The user has the Next/Previous state lists invisible when they change to Simulation mode*

As stated in the previous section, the inclusion of these lists in the Single-Step navigator was due to a constraint in Zest, with the user able to toggle their visibility. If the user has hidden the lists, the space is given over to the graphs. When the user selects to go into Simulation mode, the program checks to see if the lists have been hidden; if they have, the graphs are resized to their original position and the Simulation mode controls added to the usual area. While in Simulation mode, the menu option to toggle the visibility is disabled to prevent the user from making changes

Upon leaving the Simulation mode, the lists re-appear, meaning the user must re-select the option to hide the lists.

- *The user has one of the highlight modes on when they change Simulation mode*

The final situation considered was when the user had enabled one of the Highlight modes prior to entering Simulation mode. The decision was made

to use a form of highlighting to illustrate what was happening/had happened visually. Since a previously selected highlight mode could interfere with this representation, a check was required when entering Simulation mode to see if a Highlight mode was enabled. If one was, the mode was turned off, the graphs re-drawn to the normal colours and the menu option disabled while in Simulation mode. Upon leaving the mode, the menu option is re-enabled but the highlight mode remains off.

3.3.3.2 Simulation Run

To run the simulation, the user presses the “Play” button which causes the program to create a new *Simulation* thread. This Java thread enters a while loop, where the next transition is decided by looking at all the possible transitions and choosing one at random. This transition is then fired and an update system state process is run through, similar to the one performed when the user selects a transition. The loop then waits for a set amount of time, performs checks to see if the simulation run should end and, if it should continue, selects the next transition to fire and repeats. To stop the simulation run, the user can either use the “Stop” button - sending a signal to kill the thread, stopping the transition firing - or by setting a maximum run time/number of transitions.

Unlike the simulator in the original Workbench Debugger, the simulation mode developed for the Visual Navigator provides visual feedback to the user about what nodes have been visited and which activities have been fired. When the user initially enters simulation mode, all nodes and arcs are blank, with arcs being set to dashed lines. When a new state is transitioned to, the program assigns the nodes and arcs the correct colours for that particular system state. A check is made before assignment: if a node has previously been the current state, the darker shade of grey or blue is used; if an activity arc has previously been fired, the dashed line is replaced with a solid one. An example of a simulation run for two steps is given in Figure 3.8.

The user is able to reset the simulation at any time via the “Reset” button. When pressed, the program stops a simulation, if one is running, empties the internal lists stating which nodes have been visited/activities fired and re-draws the graphs using the normal colours (correct for the current system state) and dashed activity arcs. This allows the user to start the simulation from scratch at any time. Another feature, included in this simulation mode as it was a useful feature of the original Debugger, allows the user to perform single-stepping in simulation mode. This allows the user to step to a specific state before starting the simulation.

The use of a thread to perform the simulation calculations was necessary for two

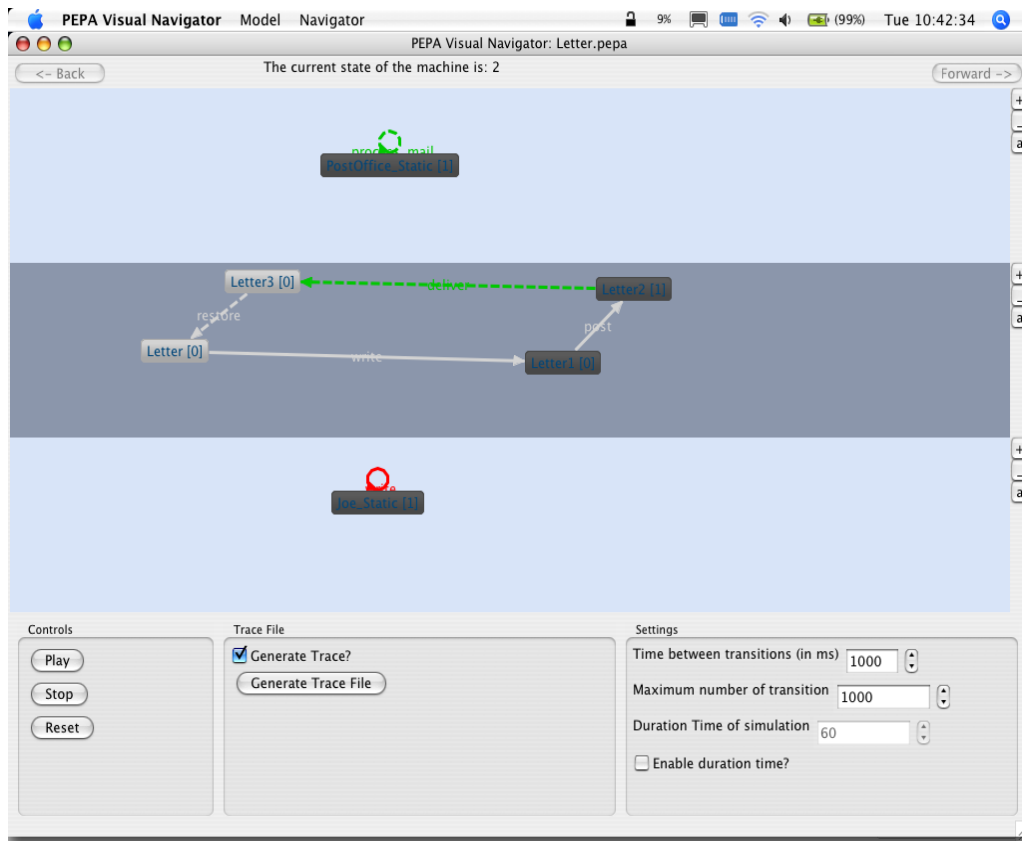


Figure 3.8: *The Visual Navigator in Simulation mode - Two transitions performed*

reasons.

- *Real-Time Updating*

To allow the visual representation to be changed after each transition fire (e.g. update the visuals of graph to reflect the new system state) it had to be possible to execute code to update the graph while still within the while loop; this was possible using the thread.

- *Allow interface access while simulation is running*

If the simulation ran in the same thread as the Visual Navigator, the user would effectively be “locked-out” of the interface while the simulator ran in the while loop. The simulator would only therefore stop when either the maximum number of transitions or maximum run-time had been reached. By having the simulator run in a separate thread to the navigator, the user still has full control of the interface (pressing the stop button, moving graph nodes, maximising and minimising etc) while the simulation is running.

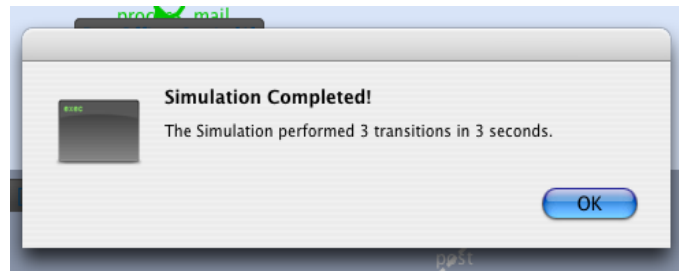


Figure 3.9: *The Visual Navigator in Simulation mode - Limit of 3 Transitions set, with 1 second between each transition*

3.3.3.3 Customise Simulation Run

To allow users to tailor the simulation run to their particular needs, the following options are made available

- *Time between transitions*

Expressed in milliseconds, the user can decide the delay between each transition fire. The default value is 1000 (1 second), with a minimum value of 0 and a maximum of 10000 (10 seconds) selectable.

- *Number of Maximum transitions*

This second option allows the user to decide how many transitions (firing of activities) the simulator should perform before stopping the simulation. By default, this value is set to 1000, with minimum and maximum of 1 and 2^{31} respectively. The default value was chosen such that the user could use the simulator for relatively small runs without needing to change the value. An example of a simulation run stopping after 3 runs (which the user has set as the maximum number) is given in Figure 3.9.

- *The Maximum Runtime*

The final option is not initially enabled by default, a check box being made available to decide whether to enable the option. When enabled, the simulator checks to see if it has run for or slightly longer than the specified maximum run time and, if it has, stops the simulation and produces a pop-up like the one in Figure 3.9 (for the example, a maximum runtime of 3 seconds would be given to have the simulation stop and produce that pop-up). The default value is set to 60 seconds and can be varied from 1 second all the way to 2^{31} seconds.

After waiting for the amount of time specified by the “time between transitions”, the simulator checks to see if the maximum number of iterations or the maximum run-time has been reached. If either of these have been reached (i.e. the first

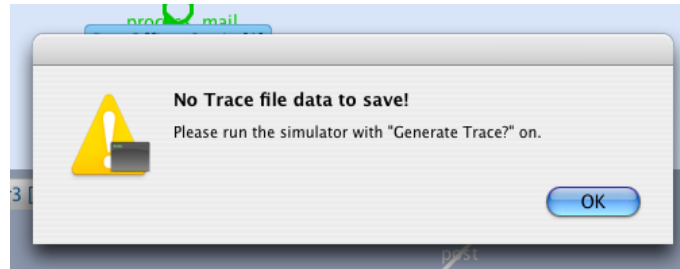


Figure 3.10: *The Visual Navigator in Simulation mode - No Trace File to Generate*

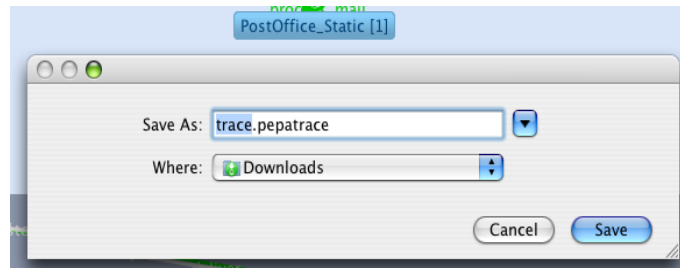


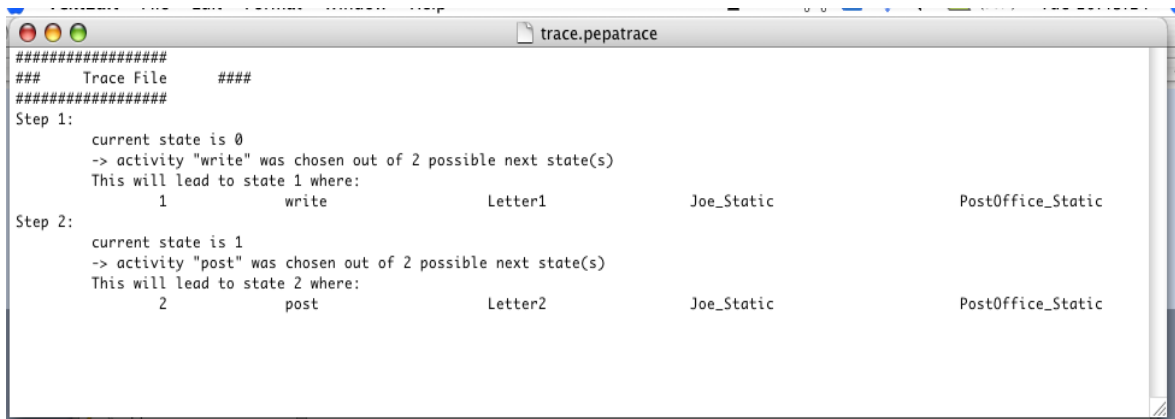
Figure 3.11: *The Visual Navigator in Simulation mode - Trace File to Generate*

to be satisfied), the simulator stops the simulation and produces the completion pop-up window.

3.3.3.4 Generation of Trace Files

The final feature available in the simulation mode is the generation of a “Trace File”. This feature was available in the Visual Debugger and produces a textual readout of the actions performed by the simulator during a run.

In order to generate a trace file, the user must first enable trace file generation by checking the “enable” check box. Attempting to generate a trace file when generation has not been enabled produces a warning dialog, as shown in Figure 3.10. Once the user has selected to enable the trace generation feature, a new empty String object is created and, for every transition fired, the information relating to that transition is appended to the String. When the user then selects the “Generate Trace File” button - either when the simulation has stopped or is still running - the save dialog window appears, as in Figure 3.11. Once the user has selected the location to save to, the contents of the String variable is written to a file by the name and in the location given. An example trace file (composed of two transitions) is presented in Figure 3.12.



```
#####  
### Trace File ###  
#####  
Step 1:  
  current state is 0  
  -> activity "write" was chosen out of 2 possible next state(s)  
  This will lead to state 1 where:  
      1      write      Letter1      Joe_Static      PostOffice_Static  
Step 2:  
  current state is 1  
  -> activity "post" was chosen out of 2 possible next state(s)  
  This will lead to state 2 where:  
      2      post      Letter2      Joe_Static      PostOffice_Static
```

Figure 3.12: *The Visual Navigator in Simulation mode - Trace File Generated*

4. Evaluation

In developing any system, an important decision has to be made on when to end the implementation phase and move onto the evaluation phase. The end of the implementation phase does not necessarily signal the end of development of the system; on the contrary, it is often the case that moving onto the evaluation stage can identify problems, potential solutions and further development for the system.

In evaluating the Visual Navigator program developed, the following topics were examined: comparing this Visual Navigator to the original Visual Debugger; testing the system; analysing which goals were met; limitations of the current build; and ideas for future development.

4.1 Comparison with the Visual Single-Step Debugger

Since the Visual Debugger was a popular and generally considered useful tool of the PEPA Workbench, the team behind the Eclipse PEPA Plug-in wanted something similar for the new tool. Although changes in the API meant development would require more than a straight port of the source code, it was hoped that the Visual Navigator could build upon the good foundations laid and expand - as well as alter - where appropriate.

Therefore, when evaluating the Visual Navigator, it seemed to make common sense to compare it to the system it was intending to replace. This comparison involved looking at how both programs visualised the PEPA models, and the features/functionality made available to the user, to see if the Visual Navigator is an improved implementation of the Visual Debugger for the new platform.

4.1.1 How They Represent the model

The representation of the PEPA models is perhaps one of the most noticeable differences when comparing the Visual Debugger and Navigator. The use of process and activity nodes to represent the PEPA model was an ideal and suitable metaphor for the processes and activities in a PEPA file. The use of large nodes for processes and small ones for activities allows for easy identification of which is which. However, representing intermediate stages with activity nodes being “current” does cause some minor confusion, as these nodes initial meaning of

“activities that cause transitions to states and not states themselves” becomes “an activity that causes a transition and a state”. This is alleviated by having the activities as strictly labeled arcs, representing transitions that cause a state change, and using separate intermediate nodes representing intermediate stages that may be occupied.

Another difference in the visual representation is the use of separate graph “areas” for each component graph in the Visual Navigator. The Visual Debugger’s use of one graph area worked well enough to represent the system, but the use of separate areas - with an alternating background colour - allows for a clearer, visual distinction of components as separate elements that may co-operate on some actions.

The labelling of the nodes and arcs varied slightly between the debugger and the navigator. In the debugger, the name of each process and activity node was placed beside the node, whereas the navigator places the process node’s name inside the node. This was an unintentional change, as this was how the Zest library placed node labels. The more significant change, however, came with the inclusion of aggregation. The workbench visual debugger did not support aggregation, meaning no numbers were assigned to nodes. The Navigator takes the aggregate value of nodes and either appends this to the name (for process nodes) or just has the number itself (for intermediate nodes) to form the label. This gives the user additional information that was not available in the workbench’s version.

A final point to make in terms of the PEPA model is the System Equation. The Workbench Visual Debugger offered no visual representation of the equation, although a “current state” label used the System Equation format and updated the component values to reflect the current system state. With the Navigator, the user is able to view the System Equation in a meaningful, graphical form without having to leave the Navigator program and view the original file.

4.1.2 Features

Although the most noticeable difference between the Navigator and Debugger comes from the representation of the PEPA model, the interface differences - and specifically the functionality offered - must be considered.

When examining the interfaces, the customisation of the graph views was the first to be considered, as this seemed to have the most differences. This was mainly due to the differences in using multiple graph areas verses one graph area. In the workbench debugger, the user could zoom in and out of the graphs (allowing, for example, one or two graphs in particular to be viewed more closely) or fit the graphs to the window (i.e. show all graphs). The zoom functions did not increase/decrease the node sizes, simply had them spaced out more. This

functionality was captured in the Visual Navigator by including a minimise and maximise button for each graph: having none selected allowed all graphs to be shown; minimising graphs could hide those not desired; maximise allowing for graphs of particular interest to have more space. The main benefit of having the maximise/minimise buttons over general zoom in/out buttons is that the user is provided with more control over what they do: they specify exactly what graphs they wish to see.

The next functionality examined was single step navigation. The debugger's method of single-clicking a green activity node to perform a transition was a very simple and effective action to represent firing an activity and performing single-stepping. The process was essentially re-used for the navigator for all but a couple of changes (double-click instead of single, to reduce chance of accidental firing; the clicking is done on the arc itself since no activity nodes). The only real change came from the inclusion of the states lists. Although only included due to a constraint of the Zest library, the lists allowed another means for user single-stepping. The benefits of including the lists may disappear when the Zest problems are fixed, which could see the lists being removed¹. If they were removed, single-step navigation would be near identical to that of the Visual Debugger.

Along with the single-step functionality, the highlight modes provide the users with something not offered in the workbench debugger: visual feedback of the history. With the workbench, the only way to know which states had been entered or which activities had been fired was to use the back button and the user noting what they saw. With the Visual Navigator, selecting a menu option instantly lets the user know exactly whether a state has been occupied or what activities have fired.

After the single-step functionality, the simulation mode was examined. The use of a separate window representation in the navigator for the simulation mode has been discussed previously. Although for the user this is a noticeable difference, further differences merit attention. The first is the highlighting of nodes visited and activities fired, a feature in the navigator missing from the debugger, allowing instant visual feedback regarding the simulation run. The second difference comes from calculating the potential next transition; the debugger takes the rate of each transition into account while the navigator gives all transitions equally weighting. The debugger's method is more sophisticated and could be used in future versions of the navigator.

The final comparison between the Virtual Debugger and Virtual Navigator's functionality was in the supported PEPA files. As mentioned before, the debugger has no support for aggregation and cannot represent multiple instances of the

¹They could be retained, it would depend upon user perception of their value.

same component (e.g. “P <> P”). This limitation means that the only way to represent multiple components is to create a new component with the same functionality. While the navigator has problems with the latter type of multiple component reference (discussed later), it does support aggregation and as a result can support those files which use aggregation to represent multiple components.

4.1.3 Overall

Overall, both the navigator and debugger share much functionality, have good points and some bad, but the navigator contains more functionality that assists the user, providing them with more control and more visual feedback than the debugger did. In addition, the Visual Navigator uses the new API used by the Eclipse PEPA Plug-in and SWT, meaning that it can be integrated easily into the new platform, whereas the Visual Debugger will only work with the Workbench.

4.2 Testing

With any piece of software, one of the most important steps in evaluating the system is the testing. There is little point in spending a large amount of the allotted development time implementing the system and additional features if no checks have been done to see whether: (a) the system performs as expected, and (b) the system is useful to the potential user.

The first check requires testing the system under normal condition (e.g. providing data or performing expected operations), extreme conditions (e.g. using the maximum values expected) and error conditions (testing situations which are considered erroneous and making sure the system handles the errors as expected). The second check involves performing user testing; having potential users use the system in a normal but, in some form, controlled manner. User testing is particularly useful for identifying potential issues and problems with a system that are not immediately apparent to those who created it.

4.2.1 System Testing

4.2.1.1 Model Representation Testing

The first set of testing examined how the program visually represented the PEPA files. This involved using PEPA files of various specifications and examining the resulting graphs.

- *normal condition test*

For the purposes of testing, a normal situation was deemed to be a user loading a PEPA file with 3 components and around 4 nodes on each. These files loaded into the navigator successfully and the colour of nodes and arcs were accurate. A couple of problems did arise however. First, activities with a number as a rate were not rendered visually; this was due to a simple omission from *PEPAData* and was corrected easily. The second issue arose when using a file with a System Equation containing “P<>P”, rendering both graphs but not assigning the state values correctly. This particular issue was not resolved and is discussed further in the “Limitations” section.

- *extreme condition test*

For extreme cases, a file with 11 components was tested, which loaded successfully and in a short amount of time. The smallest number of components supported is two, as the plug-in expects the system equation to have two components. A file with 2 components was tested and loaded fine. Large numbers of nodes seemed to produce longer load times (a “PC-LAN6” example took around 10 seconds). Additional checks identified that the model was created and instantiated in a short amount of time and the problems arose through the deriving of the state space (through the use of the supplied libraries). However, these files did load successfully.

Problems were found when attempting to load files which included activities with a choice as a target. When a file of this type is loaded, the program exits as it does not see the target (this is due to the situation not being dealt with sufficiently, see “Limitations”).

- *erroneous condition test*

The one real erroneous situation identified as part of testing involved attempting to load a PEPA file with incorrect syntax. When initially tested, the program would exit when a file of this type was loaded, printing the stack trace to the console. While exiting the program was the correct course of action, printing the stack trace did not seem the most useful manner by which to inform the user the program had done so. Instead, a pop-up window was added which is presented whenever the program cannot load the file due to problems.

4.2.1.2 Single-Step Navigation Testing

This testing section involved testing whether single-step navigation operated correctly. This involved making sure that, when the user double-clicked on an appropriate activity arc, the activity fired.

- *normal condition test*

Under normal conditions, a user would either double click on an arc coloured green or select an activity from one of the lists. For the arcs, ones that connected two different nodes were examined. The testing involved trying out the single-step navigation on several files, with the double click action being recognised and transitioning to the correct system state. Selecting the next state from the lists, this state was the one that was entered.

In order to identify whether the correct system state had been entered, the aggregation values assigned to each node were also examined. Comparing the values to those seen in the previous and current states, they were all confirmed to be correct.

- *extreme condition test*

The first extreme situation identified was when an activity arc which was a loop, with the same origin and destination node, is double clicked; the system correctly identified the state change and kept the node as a current node with the same aggregation value. The second situation tested was when multiple activity arcs had the same name, but were not synchronising ones; the system was able to correctly identify what system state should be entered when firing this activity.

- *erroneous condition test*

When considering what errors could arise during single-step navigation, the user attempting to double click on activity arcs coloured red or grey was first considered. This should not, and testing showed does not, cause any state change. These activities cannot fire, and the code correctly only allows the green arcs to cause a state change.

The second test involved testing what happened when an arc was single clicked. The interface was programmed to do nothing when a single click event occurred; the only thing that does happen is the arc is highlighted (a default property of the Zest library). Note, this highlighting is different from the “Highlight Mode” discussed previously, simply illustrating the last node or vertex clicked.

4.2.1.3 Interface Testing

Testing for the Visual Navigator interface involved testing out the different functionality available to the user to adjust the appearance of the interface and navigate through previously visited system states.

- *normal condition test*

In the normal situation, the user adjusts the appearance of the interface starting from the default setup.

- The first test involved using the maximise and minimise buttons with all the graphs initially the same size. Although some mis-calculations produced initial errors, these were corrected easily and the functionality worked correctly.
- The second test involved resizing the window; again, some initial mis-calculations caused minor errors, but these were remedied and the function worked as expected, expanding the window items correctly.
- Finally, the “toggle lists visibility” feature was tested and correctly hid/showed the lists and extended/retracted the graphs equally.

In addition, the forward and back buttons were tested by having the navigator move by a couple of states in the normal circumstance. These tests were successful and the correct states entered.

- *extreme condition test*

The first extreme situation is when the user attempts to go backwards or forwards through all the states when a large number of states have been visited (in the dozens). Test showed that the correct sequence was traversed through in both situation².

Testing the user adjustments in the extreme condition saw the following situations tested:

- *Resize the interface window with graphs maximised or minimised* (keep graphs maximised or minimise and give space to visible graphs)
- *Resize the interface window with lists not visible* (lists should still not be visible and area used for the graphs)
- *Minimise a graph while one (perhaps the same) is maximised*
- *Minimise a graph while other graphs are minimised*
- *Maximise a graph while other graphs are minimised or maximised*
- *Change graph size with lists not visible* (maximised graph or non-minimised graphs should occupy the area that was used for the list)

²If an activity that kept the system in the same state was fired several times, several instances of the system state would be traversed through. This was a design decision; including several instances of the system state instead of just one when going backwards or forwards accurately reflects the user’s history using the Visual Navigator.

- *Have hidden lists shown while graphs are minimised or maximised*
(The graphs should still be maximised or minimised but using the original graph area with the lists visible)

Although testing exposed some minor mis-calculations when re-adjusting the graphs and window area, these were corrected easily and features re-tested. This re-testing demonstrated that the functionality performed correctly and as expected³.

- *erroneous condition test*

In the erroneous situation, if the user attempts to use the back or forward buttons when they have no system states to visit. The buttons are blanked out here, so nothing happens when the user tries to press the buttons.

4.2.1.4 Highlight Mode Testing

The testing of highlight mode involved making sure the highlight modes highlighted the graphs correctly and transitioning between the modes functioned correctly.

- *normal condition test*

Under normal conditions, selecting a highlight mode when none other had been selected turns the mode on (instantly highlighting the graphs if appropriate) correctly, while turning the same mode off has the graphs return to their normal appearance. Use of the reset option correctly removes the highlight information history.

- *extreme condition test*

Under extreme cases, the user selects another highlight mode while one is currently on. This performed correctly under testing, changing to the new highlight mode.

- *erroneous condition test*

No erroneous conditions were identified for the highlight mode.

4.2.1.5 Simulation Mode Testing

The final system testing involved testing the simulation mode functionality

³Additionally, when maximising graphs while others are maximised or minimised and minimising a graph while one is maximised not only turns off the setting for the other graph(s), but deselects the respective toggle button(s) as well.

- *normal condition test*

When used under normal situations, the play, stop and reset button functioned correctly. Trace files were generated successfully after running a simulation with the trace mode enabled. All settings were adjust and functioned as expected when the simulation was run. It was noticed, however, that if the user typed in the value for a setting, the value would not be registered unless the “Enter” button was pressed. Further testing revealed that, since the value is read in and stored when the setting’s *widgetSelected* method is invoked, the value is read and stored when one of three events occurs: (i) the “Enter” button is pressed (ii) the spinner arrows beside the setting’s number box are pressed (iii) the user selects a different setting’s number box (the original setting’s number box value is then read).

- *extreme condition test*

The following checks were done for the extreme conditions:

- All settings were adjusted to their maximum and minimum values. The duration time and number of transition behaved correctly for their minimum values, while the transition time behaved correctly for it’s maximum. Although set originally to 1ms, testing showed that the transition time had a lower limit of around 200ms, possibly due to time required to perform calculations and update the screen. The maximum values for the other two settings were so large that they were not reached⁴, but ran for a long time (around 1 hour) when set and were only stopped when the stop button was used.
- The settings were changed while a simulation was run. The simulation correctly continued to run with the settings it had at the start, and when restarted used the new settings
- The reset button was pressed while a simulation was running; this stopped the simulation and reset the simulation correctly
- The generation of trace files during simulation runs was performed successfully and the correct data written
- The window size was changed during a simulation run. The window resized correctly, with all the items resized correctly. In addition, the simulation kept running the whole time.

- *erroneous condition test*

If the stop button is pressed when no simulation is running, nothing happens in the simulator (correct functionality). When attempting to generate a

⁴For reference, a time of 2^{31} ms is roughly 24 days

trace file without enabling the mode, an error window pops up informing the user that a trace file could not be generated. Attempting to adjust any settings beyond their maximum or minimum value does nothing and leaves the setting at the highest or lowest value respectively.

4.2.2 User Testing

The second phase of testing involved having actual users try out the Visual Navigator. The users were asked to comment on the general appearance of the interface and model representation, perform some simple tasks and to provide feedback on their experience.

The user testing was performed with two different users. Although this number was less than hoped for, the tests did produce some interesting feedback. The two users had different levels of experience with PEPA and the Eclipse plug-in: user “1” had some slight experience with PEPA and the Eclipse plug-in, while user “2” had no PEPA experience but had used Eclipse before.

- *User 1*

The first user understood the PEPA model presented in textual form and found the visual representation reinforced their own interpretation. The colours chosen for the states seemed intuitive to the user and they were able to perform single-step navigation and simulation easily enough. The user did state, however, that the overlapped arcs problem did hurt the visual representation but that the inclusion of the state lists was a suitable substitute.

- *User 2*

The second user was given a brief explanation of the PEPA code, including processes and the System Equation. The user was still slightly unsure but, when presented with the visual representation, stated they understood much better the make up of the PEPA file and the relationships (particularly for components and synchronising activities). The user was able to perform the tasks successfully, although had some trouble identifying the maximise button. The user suggested tooltips be added to these buttons, a feature which was added after the test. Overall, although the user did not still fully understand all the ins and outs of PEPA, the user did feel he understood it slightly better.

When asked about which highlight mode they felt would be most useful, both users stated that they felt the “Node Highlight Mode” would be most useful. Both user’s felt knowledge of which states of the components had been visited would be the most useful visual feedback, though user 2 did state that it really depended on what the particular user was interested in finding out.

When examining the feedback from the users, both stated that they felt the ability to visualise the PEPA model and perform single-step navigation and simulation helped them understand the models better. Additionally, they stated that the prospect of using PEPA in the future would be more attractive if they knew the Eclipse plug-in had the Visual Navigator tool.

4.3 Goals Met

One of the main steps in evaluating the system built is to look back at the original project goals to see if they were actually met:

- *Including the Core Functionality*

Perhaps the most important goal for the project, the system built had to provide the same core functionality as the original Workbench Visual Debugger. These included, among other things, visually representing the PEPA model, allowing single-step navigation and providing a simulation mode. The Navigator reproduces all this functionality for the user, although with some slight modification for particular functionality (such as the maximise/minimise buttons in place of the zoom and fit to window buttons).

- *Provide a new representation of the PEPA model*

The new Navigator was to use activity arcs that were labeled - removing the activity nodes - and include new “intermediate nodes” to represent stages in a transition from one process state to the next where activities still needed to fire before completion. Both these are used in the Navigator in its visual representation.

- *Aggregation Mode*

Although initially described as a “mode”, aggregation is used throughout in the Visual Navigator. If a file does not specify aggregation, the visual model uses a “1” for the current state and “0” for non-current states (an accurate representation). If the file does, the information is used to accurately assign current nodes with their appropriate aggregation value and have the number displayed.

- *Highlight Mode*

Due to some uncertainty as to which type of highlight mode would be most useful to the user, three different versions were implemented (one for the system state, one for the nodes state and one for activities fired). User testing produced some feedback on which mode was most useful, however

more user evaluation would probably better identify the most useful mode to users.

- *Visual System Equation*

A visual representation of the System Equation was lacking from the Visual Debugger. With the inclusion of this additional information, the user can see the System Equation without having to look back at the file. The visual representation also allows the user to see how the equation is structured (in terms of the hierarchy, which components co-operate directly/indirectly and on what).

- *Implement for the Eclipse PEPA Plug-in*

As a final goal, the Visual Navigator had to integrate easily into the Eclipse PEPA Plug-in. This meant doing two things, the first was using the new Eclipse PEPA Plug-in API for the backend data retrieval of the PEPA model. The second was using technology that would work with Eclipse easily for the interface and graph representations. Zest was designed for the Eclipse platform and linked in well with SWT (which was used for the interface). Therefore, the technologies used for the Navigator were ones that integrate well into Eclipse.

4.4 Limitations

Although the goals for the project were all met, the system implemented was not perfect. Aside from the problems mentioned previously, there are problems that should be addressed at some point. The solutions for these problems were not included in this build of the Visual Navigator due to time constraints.

- *Choice Node as a target*

A problem that was also present in the Visual Debugger, PEPA models that include activities with a choice as the target (e.g. “ $(a, s).((a, r).P + (b, s).Q)$ ”) will not be represented accurately. The difficulty came from deciding how to represent this in a PEPA model, the most obvious seemed to be create an intermediate node for the choice target (e.g. “ $((a, r).P + (b, s).Q)$ ”) and then evaluate each choice into the appropriate nodes and arcs. Problems also arose when deciding how to handle the possible recursion of activities with choice targets. The original Visual Debugger paper offers a potential workaround ([1], section 4.3.1 “Prefixed Choice”) but, while the same workaround works in the Visual Navigator, a proper solution would be nice and, given some time, could be implemented.

- *Multiple Instances of a Component*

If a user does not use aggregation in a PEPA file, multiple components may be included by explicitly using the component more than once in the System Equation (e.g. “ $P \langle \rangle P \langle \rangle Q$ ”). While the Visual Debugger would display just one instance of the component, the Visual Navigator can visually represent two or more instances of the component, but does not show the correct state information. Although the user sees two instances of the component, underneath there is only one instance which is edited and has values assigned, giving the incorrect representation for the system states. The best work around for this is to edit the file to use aggregation instead (e.g. “ $P[2] \langle \rangle Q$ ”). This gives the additional benefit of reducing the size of the state space produced.

- *System Equation with nesting at the same level*

This limitation is due primarily to the method used to obtain the System Equation from the PEPA model. In the method, if the System Equation has a left branch (e.g. “ $(P \langle \rangle Q) \langle \rangle R$ ”) or right branch (e.g. “ $P \langle \rangle (Q \langle \rangle R)$ ”), this branch is iterated further to obtain all the data. However, the method checks for one and then for the other, iterating on just one of the nestings if it exists. This means equations of the form “ $(P \langle \rangle Q) \langle \rangle (R \langle \rangle S)$ ” are incorrectly represented, with only one branch being created. As stated, this is a limitation in the method used to extract the information, as the *SystemEquation* object can represent an equation of this form and the graph can display it. A revision to this method would therefore be required to fix the problem (possibly one that used a recursive approach over the sub-trees instead of an iterative one).

- *Current build requires deriving the state-space*

Although using code from the PEPA Eclipse Plug-in API, the current build has not yet been fully integrated into the plug-in: the Navigator runs as a standalone program, importing the necessary packages from the API. This means the program needs to derive the state-space itself in order to obtain the information. Once fully integrated, the Visual Navigator could be configured such that the user has to select to derive the state-space for the current file before performing visual navigation, the Visual Navigator then using this derived state-space instead of deriving it's own

- *Button overlapping when number of components is large*

A problem identified involved the maximise, minimise and hide activity arc buttons: if a sufficiently large number of component graphs are required to be displayed in the window, very small graphs are made and result in the buttons overlapping one-another. A solution for this problem would come

in two forms: first, make the minimum height for a graph the same height as the three buttons added together; then make the area of the window assigned to graphs scrollable, so that all the graphs could be displayed and only require the user to scroll to them.

- *Save Current Layout*

A feature found in the Visual Debugger allowed the user to save the current layout of nodes of the visualisation to an XML file. When the user later loaded the same PEPA file, this layout could be loaded to re-create the previous set-up. The functionality would be very useful to include as part of the Visual Navigator, but was not implemented at this time due primarily to the time-constraints (the implementation of this functionality would require an XML parser for the file format, determining the format and devising a way to obtain/restore the node placement information)

4.5 Future Improvements

The final part of the evaluation stage involved examining the system made and - looking at features available and not available plus suggestions made during user testing - seeing what could be added or expanded upon for future versions. Most of the future improvements are as a result of problems raised earlier and have already been discussed; the following list includes features that do not address specific problems with the current system but rather could improve the user experience:

- *On-Line Help Option*

All programs have a learning curve associated with them. This learning period could be eased by the inclusion of an on-line help option. The Help menu option could include: detail about the general window set-up; what graphs/colours represent; what functionality is available and how to use them; an on-line tutorial and much more.

- *A Menu Bar for each Graph*

In the current implementation, the maximise, minimise and hide arc buttons are placed beside the graph. A possible change could see a menu bar included on top of each graph with these buttons aligned horizontally, similar to a window with its window bar. When a graph was minimised, instead of hiding the graph completely - with only the buttons at the side indicating which graph is minimised - the graph could be hidden, with the space used for the remaining graphs, and only this "title bar" with the buttons visible.

- *Removal of the Next/Previous State Lists*

The inclusion of these lists was due to a constraint in the current build of the Zest libraries. A decision will have to be made about the usefulness of keeping the lists when a future release of Zest which addresses this constraint is released: either remove them as the only purpose they served has been remedied, or keep them because they have proved to be useful beyond their original purpose.

- *Allow the user to choose the state colour scheme*

For the purpose of consistency, the colour scheme used in the Visual Debugger was reused for the Visual Navigator. However, the colours may not be appropriate for all users, particularly those with Red-Green colour blindness (or, for that matter, any type of colour blindness). Allowing the user to customise the colour scheme could both alleviate this problem and allow the user to use a more meaningful colour scheme for them.

5. Conclusion

The overall goal of the project was to create a Visual Navigator for the Eclipse PEPA Plug-in. The program had to:

- *provide the core functionality of the original Workbench Visual Debugger*
- *use the new API of the Eclipse PEPA Plug-in*
- *use graphics libraries that would integrate easily into Eclipse*

The first of these goals was met by the program. The user is able to load a PEPA file - including those with aggregation, which were unsupported in the Visual Debugger - into the visual navigator to produce a graphical representation of the model. The user can then perform single-step navigation and simulation running. The Visual Navigator also offers additional features to the user, not present in the original Workbench Visual Debugger.

The second goal was achieved by using this API to obtain the information about the PEPA model; both the model and state space information.

The final goal was achieved by using the SWT to build the interface (the native widget toolkit of Eclipse) and Zest for the graphs (which was designed for the Eclipse platform and integrates well into SWT).

Although meeting these goals, it is possible to improve upon the Visual Navigator. Constraints from the current build of Zest, coupled with limitations in the interface implementation, do cause some problems when rendering certain PEPA files.

However, the current build of the Visual Navigator is still very much usable and can be included as part of the Eclipse Plug-in.

Furthermore, tests carried out as part of this project have shown that, when presented with the Visual Navigator, novices to the PEPA language found it much easier to understand the model and language alike. Considering one of the hopes for the Eclipse PEPA Plug-in project was to expose more people to the benefits of the language, the inclusion of this Visual Navigator is likely to increase the adoption of it.

Bibliography

- [1] *The Needham Lecture, Tuning Machines: From Composition to Performance*, Jane Hillston, 3rd May 2005
<http://www.dcs.ed.ac.uk/pepa/tuningsystems.pdf>
- [2] *Finite State Machine Visualiser for Discrete Event Simulation, A Visual Debugger for the PEPA Workbench*, Antony Rinaldi, 2nd March 2005
<http://www.dcs.ed.ac.uk/pepa/visualsinglestepdebugger.pdf>
- [3] *Learn Eclipse GMF in 15 minutes*
<http://www-128.ibm.com/developerworks/opensource/library/os-ecl-gmf/>
- [4] *The Eclipse Graphical Editing Framework (GEF)*
<http://www.eclipse.org/gef/>
- [5] *The Eclipse Graphical Modeling Framework (GMF)*
<http://www.eclipse.org/gmf/>
- [6] *SWT: The Standard Widget Toolkit*
<http://www.eclipse.org/swt/>
- [7] *Zest: The Eclipse Visualization Toolkit*
<http://www.eclipse.org/mylyn/zest.php>
- [8] University of Edinburgh *Modelling and Simulation* course lecture notes, Jane Hillston, September 2007
<http://www.inf.ed.ac.uk/teaching/courses/ms/notes/Notes.ps>

Appendix A. Algorithms

Algorithm 1 Extract Transition Information

```
1: for each process definition child of “PROCESSES” do
2:   Get process name, this identifies the origin of the transition
3:   Get associated equation for process (describes all possible transitions from
   this state)
4:   if a PrefixNode then
5:     Equation has just one transition (e.g. “(a,r).Q”)
6:     Call method to return the transition as a vector of Strings (e.g. <“(a,
   r)”, “Q”> )
7:     append origin name to vector of Strings representing the transition (e.g.
   <“P”, “(a, r)”, “Q”> )
8:     pass as a parameter to the getTransition method, which takes a vector of
   Strings representing a transition and creates the corresponding Activity and
   IntermediateNode objects necessary to represent the transition in
   PEPAModel
9:   else if a ChoiceNode then
10:    there is more than 1 transition, need to evaluate left hand side then right
11:    if Left-Hand side a ChoiceNode then
12:      call method that takes the node and returns a list of PrefixNodes
13:      for Each PrefixNode in the list do
14:        // Same steps as stated before for a PrefixNode
15:      end for
16:    else if Left-Hand side a PrefixNode then
17:      // Same steps as stated before for a PrefixNode
18:    end if
19:    if Right-Hand side a ChoiceNode then
20:      // Same as for left-hand side
21:    else if Right-Hand side a PrefixNode then
22:      // Same as for left-hand side
23:    end if
24:  end if
25: end for
```

Algorithm 2 getTransition(Vector of Strings)

```

1: if Vector of Strings is less than three then
2:   /* if less than three items, does not contain the “process” “activity” “pro-
   process” needed for a transition, so exit */
3:   return Error
4: else if Vector of Strings has exactly three Strings then
5:   /* The vector of strings describes a transition that has just one activity
   that causes a transition from one process to another - no intermediate nodes
   are required */
6:   Create a new Activity object where:
7:   Name of activity: the name in the activity
8:   Rate of activity: Rate object in list with same name as the rate in the
   activity String
9:   Activity origin node: ProcessNodeVis with same name as first string in
   vector
10:  Activity destination node: ProcessNodeVis with same name as last string
   in vector
11:  return
12: else
13:  /* The vector of strings describes a transition that has two or more activities
   that cause a transition from one process to another - intermediate nodes
   are required */
14:  Create a new Activity object using the name and rate given by the first
   activity (second item in the vector), using the ProcessNodeVis of the first
   entry as the origin node.
15:  For the destination node, pass the remaining entries of vector that ap-
   pends them into a string representing an intermediate state (e.g. <
   “(b,r)”, “Q” >⇒ “(b,r).Q” ). Either get or create an IntermediateNode
   with this name, set it as the destination node.
16:  for the second activity, until the last activity in the list do
17:    Create new Activity object, using the name and rate given by this current
   activity
18:    use the last IntermediateNode as the origin node
19:    set the destination as the IntermediateNode composed from the remain-
   ing items in the vector
20:  end for
21:  /* Once here, the final activity is the n-2 entry in the vector (for length n).
   Therefore, the destination node is the final entry i.e. the target process of
   the transition*/
22:  Create new Activity object, using the name and rate given by this final
   activity
23:  use the last IntermediateNode as the origin node
24:  set the destination as the ProcessNodeVis with the name of the last entry
   in the transition vector
25: end if

```

Algorithm 3 systemEquation(ModelNode model)

```

1:  $id \leftarrow 0$  // id of current co-operation activities
2:  $parentID \leftarrow -1$  // parent ID, initially -1 for the root
3: Take "ModelNode" and extract the "System Equation" node
4: Create a list of "SECooperationAction" objects
5: Create a list of "SEComponent" objects
6: if the child of the "System Equation" node is a "CooperationNode" then
7:   while Left co-operation flag true OR Right co-operation flag true do
8:     // First: get list of all the activities
9:     for Each activity in the activity child do
10:      get activity name and add to list of Strings
11:    end for
12:    Create a new "SECooperation" object using the list of activities, the current id and parent id value. Add this to the list
13:    // Second: inspect left child
14:    if Left hand child a CooperationNode then
15:      // A nesting, more than 2 components
16:      set flag to indicate co-operation in left subtree
17:    else if Left hand child a ConstantProcessNode then
18:      // A component (e.g. "P")
19:      set flag indicating left subtree to false
20:      create new "SEComponent" object using the name given by the node, set the aggregation value to 1 and use the current id
21:    else if Left hand child a AggregationNode then
22:      // A component with an aggregation value assigned (e.g. "P[4]")
23:      set flag indicating left subtree to false
24:      create new "SEComponent" object using the name given by the node, use the aggregation value also given by the node and use the current id
25:    end if
26:    if Right hand child a CooperationNode then
27:      // A nesting, more than 2 components
28:      set flag to indicate co-operation in right subtree
29:    else if Right hand child a ConstantProcessNode then
30:      // A component, similar left hand side
31:    else if Right hand child a AggregationNode then
32:      // A component with an aggregation value assigned, similar left hand side
33:    end if
34:    if Right hand sub tree is true then
35:      Extract the left, right and action children, use them for the next iteration
36:    else if Left hand sub tree is true then
37:      Extract the left, right and action children, use them for the next iteration
38:    end if
39:     $parentID \leftarrow id$ 
40:     $id++$ 
41:  end while
42: end if
43: Create a new SystemEquation object using the two lists of SECooperation and SEComponent objects.

```
